Dealing with data with plain Python

⚠ **NOTE**: Code in this document has been tested with Python 2.7.7. However, it may work also with previous and newer versions.

Tabla de contenido

0. Preliminaries

Interacting with the Python interpreter

The Python interpreter allows you to write Python code interactively, that is, you write a sentence and it gets executed immediately. You access the interpreter shell by simply invoking it from your command prompt.

```
[anaconda@localhost Desktop]$ python
Python 2.7.7 |Anaconda 2.0.1 (64-bit)| (default, Jun  2
2014, 12:34:02)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-54)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and
https://binstar.org
>>>
```

In the snippet above, you can see the version (2.7.7) – which by the way is important – and some other details of the interpreter. In our case, the interpreter is provided in the Anaconda package, a distribution of Python libraries.

⚠ **NOTE**: The snippet shows it is using a C implementation of Python, that is, CPython. There are other implementations, for example, one on Java bytecode (Jython). In principle, this does not matter much as the language is the same.

This is extremely useful when learning Python or testing something quick. We will be using the Python interpreter, even though this is just for starting.

⚠ **NOTE**: IPython is an enhanced Python interpreter for data science work that allows for a mix of interactive execution and documenting the work done with the concept of *notebooks*.

If we want to get out of the Python interpreter:

```
>>> quit()
[anaconda@localhost Desktop]$
```

Printing something

Expressions are evaluated interactively.

```
>>> 2+2
4
>>> 2+2, 5+3, "I love Python"
(4, 8, 'I love Python')
```

```
>>>
```

And you can print also expressions with the print sentence.

```
>>> print 2+2, "I love Python"
4 I love Python
```

NOTE: This is one of the things that changed when moving to Python 3.x. Print becomes a built-in function and you have to use parentheses: print(…)

The interpreter has memory, and you can use variables to “remember” values (more on variables later).

```
>>> a = 2+2
>>> print a
4
>>> a = a +1
>>> print a
5
```

You can do the same if you save the code to a file, then execute it either:

- From inside the interpreter prompt, e.g.
  `>>>execfile('myfile.py')`

- Or directly when invoking the interpreter, e.g.
  `$ python myfile.py`

The interpreter tries to be helpful

As humans, we do not always do everything right, so we not always write correct code. A very important skill to be developed is that of understanding and getting insights from error messages.

```
>>> print b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Here we try to print variable b, but we never used it before, so it does not exists yet. The interpreter says that b “is not defined”, and what we have here is an exception, that is, the line of code entered could not be executed and the interpreter complained.

Errors are sometimes more informative or less. But in any case, they give us some hint on what happened.

```
>>> print 2+2, "I love Python
  File "<stdin>", line 1
    print 2+2, "I love Python
                            ^
```

```
SyntaxError: EOL while scanning string literal
```

In this case, the caret in below the sentence points to the particular column in which the process was broken. And in this case, the exception is not a `NameError` as before, but a `SyntaxError`.
It is apparent that the problem is there is a missing quotation mark, but the description of the error is a bit more obscure. It says that it found End of Line (EOL) while it was attempting to find the trailing quotation mark, that is, it was "scanning a string literal".

TRICK: In many cases, you can Google exact error messages as "SyntaxError: EOL while scanning string literal" and find some hints (maybe not for so simple typographical errors, but it is useful for less obvious ones).

You can use more than one-line sentences. Here comes an example.

```
>>> socrates_is_human = True
>>> if socrates_is_human:
... print "socrates is mortal"
  File "<stdin>", line 2
    print "socrates is mortal"
        ^
IndentationError: expected an indented block
>>> if socrates_is_human:
...    print "socrates is mortal"
...
socrates is mortal
>>>
```

Once the interpreter detects we are attempting a multi-line sentences (`if` sentence), it does not immediately execute the line, but waits for more input from you. However, you are responsible for correctly indenting your code. Indentation in Python is mandatory to identify code units.

1. Numbers

Numbers have types

Integer literals can be represented as such and are subject to common arithmetic operators:

```
>>> (2+3)*2
10
>>> type(2+2)
<type 'int'>
```

Plain integers are limited in their maximum value; long integers have no limitation (except that of the memory itself). Notice the trailing "L" in the following paste:

```
>>>from sys import maxint
>>> maxint+1
9223372036854775808L
>>> type(maxint+1)
<type 'long'>
```

Python provides double precision numbers. According to the language spec: "*You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow*".

```
>>> type(3.1416)
<type 'float'>
```

You can mix integer and float numbers, and types promote, that is, they are implicitly converted to the more reasonable choice.

```
>>> 2.0* 3
6.0
>>> type(2.0*3)
<type 'float'>
```

Beware of precision

Numbers in Python have an internal representation. In case of decimal numbers, there are some caveats about precision that you need to consider.

```
>>> a=1.0/10
>>> a
0.1
>>> a + 0.2
0.30000000000000004
>>> a + 0.2 == 0.3
```

```
False
```

The problem comes because the internal representations of decimal numbers are actually approximations. This will probably cause problems if you depend on decimals for taking decisions in your program.

You can explicitly round numbers for your required precision with the `round` built-in function.

```
>>> a = 0.1 + 0.2
>>> a
0.30000000000000004
>>> a = round(a, 2)
>>> a
0.3
```

NOTE: See more built-ins at: https://docs.python.org/2/library/functions.html

An alternative if we need decimal arithmetic is using the `Decimal` type that is in the `decimal` module. Following the example:

```
>>> from decimal import Decimal
>>> Decimal(0.1)+Decimal(0.2)
Decimal('0.3000000000000000166533453694')
>>> Decimal("0.1")+Decimal("0.2")
Decimal('0.3')
```

KNOW MORE: The problem is that decimal numbers are represented in binary form internally, and not every decimal fraction can be represented exactly as a binary fraction.
See: https://docs.python.org/2/tutorial/floatingpoint.html

Expressions have types also

```
>>> 1/3
0
```

What?

```
>>> type(1/3)
<type 'int'>
```

Ok, that's integer division.

NOTE: This is one of the things that changed when moving to Python 3.x!

Exceptions and numbers

This may be obvious.

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

But it may happen if we do not test numeric variables when dividing.

There is a way to deal with infinity(inf) and not-a-number (nan) values. For example:

```
>>> a = float("nan")
>>> a
nan
>>> from math import isnan
>>> isnan(a)
True
>>> b=2
>>> isnan(b)
False
```

The meaning of Inf and NaN follows the conventions of IEEE 754 standards (see http://en.wikipedia.org/wiki/IEEE_floating_point).

KNOW MORE: In addition to all the above, it is really useful to take a look at the functions in the math package: https://docs.python.org/2/library/math.html

2. Variables

Variables in Python are names that reference portions of memory that store a value. With basic numeric data types, we can think on variables as “boxes” and the assignment operator “copies” values from one box to the other. For example:

```
>>> a = 10
>>> b = 20
>>> b = a
>>> print b
10
>>> b = b +1
>>> print b
11
>>> print a
10
```

However, that semantics of “boxes” and “copies” do not hold for every data type, and we must be aware of that.

Now let’s do a small exercise: interchanging the value of two variables (“swap” the values).

```
>>> a=1
>>> b=2
>>> a=b
>>> b=a
>>> print a
2
>>> print b
2
```

Not what we intended!. The problem is that when we copy a variable onto other, the previous value gets “lost” or “overwritten”.

We need a “helper” variable to do the copy.

```
>>> a=1
>>> b=2
>>> aux=b
>>> b=a
>>> a=aux
>>> print a
2
 >>> print b
1
```

LET'S VISUALIZE: From here on, it is better to visualize the contents of the memory in Python to understand what's going on. For that we have this useful tutor:
http://pythontutor.com/

3. Functions – the basics

Functions are a way of grouping code in Python, so that you can reuse the same function as many times as you want.

For example, the **break-even point of a company** is calculated to find out the amount of sales required to cover its expenses. It's calculated using the following formula:

Break-even point (BEP) in unit sales = total fixed costs / (sale price – variable cost)

The code for computing that is as simple as a sentence:

```
bep = total_fixed_cost / (price – variable_cost)
```

However, it would be useful to use it many times without a need to repeating that code. This is where functions come, e.g.:

```
# Some financial utility functions

# Computes the break-even point in unit sales
def bep(total_fixed_cost, price_per_unit, variable_cost):
  return total_fixed_cost /(price_per_unit - variable_cost)

# Some examples
tfc = 10000
price = 250
production_cost = 124
print "You have to sell "+ str(bep(tfc, price,
production_cost))+ " units"

# That was too ambitious, let's try to reduce fixed costs
tfc = tfc * 0.2
breakeven = bep(tfc, price, production_cost) print "You have
to sell "+ str(breakeven)+ " units"
```

The names used for the values in the function definition are formal parameters, used inside the function. These are mapped to the actual parameters in each call to the function.

Looks good, but when we execute this we have:

```
>>> execfile("financial.py")
You have to sell 79 units
You have to sell 15.873015873 units
```

Multiplying by 0.2 has made the tfc variable a float, then all the operations promote to float. However, it does not make sense to express units with decimals, so a workaround could be changing the code of the function to:

```
def bep(total_fixed_cost, price_per_unit, variable_cost):
  aux = total_fixed_cost /(price_per_unit - variable_cost)
  return round(aux)
```

And then:

```
>>> execfile("financial.py")
You have to sell 79.0 units
You have to sell 16.0 units
```

MORE ON PARAMETER PASS: You can also pass parameters matching the parameter names in the function and not respecting the position in the list of parameters. For example:
b = bep(tfc, variable_cost = production_cost, price_per_unit = price)

Let's build a module

Modules are intended to reuse code. You have actually built a module simply by separating some functions in the financials.py file.

If we want to use the functions in our module from other file, we can try to simply call it:

```
# This is in file example.py
print bep(1000, 50, 20)
```

But when we execute it, we'll find this error:

```
[anaconda@bigdata1 Desktop]$ python example.py
Traceback (most recent call last):
 File "example.py", line 1, in <module>
   print bep(1000, 50, 20)
 NameError: name 'bep' is not defined
```

Python needs to know where to locate the bep function. The import sentence is precisely used for this purpose.

```
# This is in file example.py
from financial import bep
print bep(1000, 50, 20)
```

Parameters are copies!

Now let's go to the tricky part of parameter passing and scope. Let's try the following file.

```
a= 20
def f(a):
   a = a +2
   return a
print a
```

```
f(a)
print a
```

This will result in:

```
[anaconda@bigdata1 Desktop]$ python scope.py
20
20
```

Note that the a variable outside the function is not modified by the function when executed. The two “a” are in different scopes!

Also, the value of a passed to f is a copy of the original one. If you try to write the swap functionality in a function you will get a surprise:

```
def swap(a,b):
   aux = b
   b = a
   a = aux

x= 5
y= 2
swap(x,y)
print x, y
```

The values of x and y have not changed. As they are copies, the originals are intact. The only way to do this using a function is returning pairs of values swapped:

```
def swap2(a,b):
   return b,a
```

4. What if…

Almost any non-trivial program contains conditional code, that is, code that should be executed only on certain circumstances. Let's see a very simple example.

The net present value (NPV) is an indicator of the value that an investment brings to the firm. The decision depends on NPV being zero, negative or positive. A typical decision logic can be the following:

```
>>> npv = 2000
>>> if npv > 0:
...     print "Go ahead"
... elif npv==0:
...     print "Indiferent"
... else:
...     print "Dont go"
...  Go ahead
```

The last line is the response to the if sentence.

In some cases, a sequence of conditionals is needed. For example, in the following:

```
# Computes startup evaluation in M$ based on the ideas on:
# http://berkonomics.com/?p=1214
def berkus_eval(idea, prototype, team, partners, sales):
   plus = 0.5
   value = 0
   if idea:
      value +=plus
   if prototype:
      value +=plus
   if team:
      value+=plus
   if partners:
      value+=plus
   if sales:
      value+=plus
   return value
```

Logical values

In some cases, functions may return logical values that can be used further. For example, in order for a company to be listed in the NYSE it needs to have a minimum price of $4. So, the decision is as follows.

```
# Decide if a company can be listed at NYSE
```

```
#based on price per share (in $):
def can_be_listed_NYSE(price):
   return price >=4;
```

Then:

```
print can_be_listed_NYSE(3.8)
if can_be_listed_NYSE(5.2):
   print "It can be listed"
else:
   print "Can't be listed"
```

And this will display:

```
False
It can be listed
```

Calling functions from functions

This is an excerpt of the definition of SMEs by the European Comission:

The main factors determining whether a company is an SME are:

- **number of employees** and
- either **turnover** or **balance sheet total**.

Company category	**Employees**	**Turnover**	or	**Balance sheet total**
Medium-sized	< 250	≤ € 50 m		≤ € 43 m
Small	< 50	≤ € 10 m		≤ € 10 m
Micro	< 10	≤ € 2 m		≤ € 2 m

So for example, a small company has less than 50 employees and either a turnonver no larger than 10 m or a balance sheet total no larger than 10 m. Obviously, **this definition tacitly entails also that the company is not micro**!.

Then to solve this problem we can do it in two steps:

- Define a function to determine if a company is micro.
- Then reuse that function in another function to determine if the company is small.

Here goes the code.

```
# Defines function to determine
# the kind of enterprise according
# to official EU definitions.

# Turnover and balance sheet total in MEuros
def ismicro(employees, turnover, balance):
   return (employees<10) and (turnover<=2 or balance<=2)

def issmall(employees, turnover, balance):
   if ismicro(employees, turnover, balance):
      return False
   else:
      return (employees<50) and (turnover<=10 or
balance<=10)
```

And if you execute the following:

```
from company_types import ismicro, issmall

print ismicro(5,2,2)
print issmall(5,2,2)
print issmall(12,2,2)
```

You will get:

```
True
False
True
```

5. Handling text

Strings are sequences

Strings are sequences of characters. String literals can be enclosed in single or double quotes. Strings are a type of sequence object in Python (as lists and tuples). In this case, it is a sequence of characters. All sequences can be indexed, sliced and concatenated. Let's see an example.

```
# Manipulate tweets

tweet = """We'll be matching donations made
to @pypyproject this month up to
$10,000 total! Go donate! """

print len(tweet)

print tweet[6]

print tweet[9:14]

tweet = tweet + "Donate!"
```

Also, if you try to access a character that is out of the string length, e.g.:

```
print tweet[251]
```

You will get an indexing error:

```
IndexError: string index out of range
```

All the operations on strings so far are the same for all sequences in Python. But strings have specific "methods" also, that are invoked with dot notation. If we want to know the methods of an object we can do it as follows.

```
>>> s = "Hello"
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

And then to know about a particular method:

```
Help on built-in function replace:

replace(...)
    S.replace(old, new[, count]) -> string

    Return a copy of string S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

Strings are immutable

Strings are immutable, which means that you cannot change its components, in this case, you cannot change its characters.
So if you try the following:

```
tweet[6] = ‘X’
```

You will get a type error:

```
TypeError: 'str' object does not support item assignment
```

When we concatenated with the plus operator above, a new string object was created internally.

Strings go international

Python strings are actually ASCII based, that is, internally represented using the 8-bits and the ASCII code. You can get the ASCII numbers using the ord() function.

```
>>> s = "El niño es alemán"
>>> print s
El niño es alemán
>>> ord(s[4])
105
>>> ord(s[5])
195
```

The character “ñ” is actually not original ASCII, and may not print well in other systems, it depends on the localization of the operating system.

If we try to do the following:

```
>>> s = "Before " + chr(258) +"after"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: chr() arg not in range(256)
```

It complains as it is out of the 8-bit scope.

In Python 2.x, there are actually two types, str and unicode to represent strings. We have already dealt with str in the examples. If we want to handle Unicode we need to do it explicitly:

```
>>> s = u"El niño es alemán"
>>> type(s)
<type 'unicode'>
```

Text to numbers and back

This is simple, you only need to use the str and int builtin functions:

```
>>> a = 7.2
>>> s = str(a)
>>> s
'7.2'
>>> s = s + "5"
>>> a = float(s)
>>> a
7.25
>>>
```

6. Data structures

The built-ins

The power of Python is in the data structures. But which and when to use them? Let's look first at some differences. This table is for reference, examples come below.

	Homogeneous?	Mutable?	Duplicates?	Order?
list	Yes (typically)	Yes	Yes	Yes
tuple	No (typically)	No	Yes	Yes
set	Yes (typically)	Yes	No	No
frozenset	Yes (typically)	No	No	No
dict	No (typically)	Yes	Yes (values)	No

Set elements need to be hashable! This makes them more efficient for some implementation reasons that we do not need to cover here.

MORE: In the standard library, check the *collections* module: https://docs.python.org/2/library/collections.html

Here we are covering lists only, which are useful for most situations dealing with numerical or string data.

Lists versus tuples

Tuples are immutable, so assignment is not allowed.

```
>>> a = (2, 3, 5, 12, 22, 5)
>>> len(a)
6
>>> a[2]
5
>>> a[2]=0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a + (3, 2)
(2, 3, 5, 12, 22, 5, 3, 2)
```

Lists are mutable objects, they can be dealt with as with other sequences:

```
>>> L = [2, 3, 5, "a"]
>>> len(L)
4
>>> L[1]="b"
>>> L
[2, 'b', 5, 'a']
>>>
```

You can get lists with some builtin functions, and tuples as a return of some functions.

```
>>> import time
>>> a = time.localtime()
>>> print a
time.struct_time(tm_year=2014, tm_mon=10, tm_mday=12,
tm_hour=17, tm_min=53, tm_sec=32, tm_wday=6, tm_yday=285,
tm_isdst=1)
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The first one, a tuple, is a sequence in which position has semantic value. The first position is always a year. This tuple functions as a lightweight record or struct.
The second one, a list, is a sequence where we may care about order, but where the individual values are functionally equivalent.

Manipulating lists

There are a number of list-specific functions available.

```
>>> a=range(10,50,5)
>>> a
[10, 15, 20, 25, 30, 35, 40, 45]
>>> a.append(50)
>>> a
[10, 15, 20, 25, 30, 35, 40, 45, 50]
>>> a.reverse()
>>> a
[50, 45, 40, 35, 30, 25, 20, 15, 10]
>>>
```

Structures can be nested also. Using lists, this allows us to represent matrices.

```
>>> m = [range(1, 10, 2), range(10,1, -2), range(20, 30, 2)]
>>> m
[[1, 3, 5, 7, 9], [10, 8, 6, 4, 2], [20, 22, 24, 26, 28]]
>>> m[1][1]
8
>>> m[2]
[20, 22, 24, 26, 28]
```

```
>>>
```

You can get individual elements using two times indexing, or a row indexing one time.

Names are references – be careful with mutables

An important concept (but somehow confusing for beginners) is that variables are references to objects. This has as a consequence for mutable objects (as lists) that several variables. Let’s see an example.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a.append(4)
>>> b
[1, 2, 3, 4]
>>> b[1]=0
>>> a
[1, 0, 3, 4]
>>> b = None
>>> a
[1, 0, 3, 4]
```

Mutable objects can be changed inside functions!

Even though it is not very common, mutable objects as lists can be changed inside functions (unlike basic data types like integers).

```
def change(li):
   del li[len(li)-3]
   li[len(li)-1] = 0

S = [1200, 1000, 800, 100, 10000]
change(S)
print S
```

The first sentence removes the element at position 5-3, that is, the number 800. Then it sets to zero the last of the remaining elements. This will result in:

```
[1200, 1000, 100, 0]
```

7. Looping on data

Loops (iterative sentences) allow us to apply a fragment of code repeatedly. When dealing with data, this is often used to apply that code to each element of a sequence, e.g. to all the elements of a list. This is often called a **iteration**.

An example of such kind of processing is the following:

```
>>> a = range(0,100,10)
>>> a
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> for item in a:
...    print item**2
...
0
100
400
900
1600
2500
3600
4900
6400
8100
```

We can also take sequences as parameters in functions, then returning new sequences based on them. An example follows.

```
def grade(scores):
   grades=[]
   for score in scores:
      if score >= 5:
         grades.append("Pass")
      else:
         grades.append("Fail")
   return grades


scores = [0, 5.0, 5.5, 7, 4, 5, 3]
g = grade(scores)
print g
```

This will result in:

```
['Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass', 'Fail']
```

8. Some magic to get data

Now that we have been doing the basics, it is time to get hands on with getting data. We will do it as simple as possible, taking advantage of Yahoo Finance.

Yahoo Finance has a facility to retrieve stock quotes by simply issuing an HTTP request. This can be done with a single sentence using the urlretrieve method of Python's urellib module.

```
from urllib import urlretrieve

where_the_file_is = 'http://real-
chart.finance.yahoo.com/table.csv?s=IBM&f=2014&c=1962&ignore
=.csv'
urlretrieve(where_the_file_is, 'stock_data.txt')
```

The magic is in the URL. It has GET values for the symbol ("IBM") and then a pair of dates.

It will get a file with the following structure:

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-10-10,185.86,187.74,185.10,185.93,5090200,185.93
2014-10-09,189.12,189.50,186.09,186.42,2625400,186.42
2014-10-08,185.97,189.60,185.61,189.36,2984800,189.36
…
1962-01-03,572.00,577.00,572.00,577.00,288000,2.48
1962-01-02,578.50,578.50,572.00,572.00,387200,2.45
```

This is the typical data structure in which "columns" are separated by commas.

9. A little bit of data processing

Now let's do the most minimalistic data wrangling operations with the file we have downloaded from Yahoo Finance.

The first task is loading data from the file. This can be done very easily in Python with the built-in file handling capabilities. The idea is to get a piece of the data (in this case only the adjusted close column) and create a list with that data. Here goes the code:

```
# Example data manipulation with
# Yahoo Finance data.
# Note: assumes the header line has been removed

def read_data(filename):
   f = open(filename, 'r') # open for reading
   close_values = []
   for line in f:
      close_values.append(get_close(line))
   return close_values

def avg(l):
   aux=0
   for i in l:
      aux = aux + i
   return aux/len(l)

def get_close(l):
   fields = l.split(',')
   last = len(fields)-1
   return float(fields[last])

close = read_data('stock_data.txt')
print close[0]
print close[1]
print len(close)
print max(close), min(close), avg(close)
```