Data Structures and Algorithms with Python - Tutorial 2

September 13, 2016

This tutorial will continue introducing a few more important Python concepts. Most importantly, you'll first learn how to deal with errors and exceptions and the basics of debugging code. This is the core part of the tutorial. In the latter parts of the tutorial, you'll have the chance to explore more of Python's built-in data types: tuples, dictionaries, and sets. Finally, you'll learn how to use Python to read information from files.

But let's start with an exercise to recap the idea of how we use functions.

0.1 Exercise: scope of a function

Consider the following code:

```
x = 10

def add(a):
    x = x + a
    print(x)

add(2)
```

What happens when you call `add(2)`? Why?

Change your function to:

```
def add(a, b):
    c = a + b

add(2, 3)
print(c)
```

What happened now? What do you need to add to get a value from the function?

0.2 Spoiler alert

Functions come with a **context**: when you enter the function `add`, only the parameter you passed it (in this case, `a`) are defined. Similarly, the variables you define inside a function **do not** exist outside that function.

1 Handling errors in Python: try/except

Python is expressive in the way it deals with errors. This, combined with its interpreted nature make it easy to debug incorrect code. Here we will first look at how errors are handled in Python using the try/except structure then we will look at how to debug incorrect programs.

1.1 Python errors:

In Python, when an error occurs, the interpreter tries to give you as much information as possible about the error. For instance, consider the following code:

```
>>> variable = 'hello'
>>> print(varialbe)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-60ea18b6f15e> in <module>()
----> 1 print(varialbe)

NameError: name 'varialbe' is not defined
```

The first line tells you that your instruction `print(varialbe)` returned a NameError. Fairly explicit. Then the *Traceback* is added, i.e. the lines at whitch the error occured. Finally, details about the `NameError` is given, here, `name 'varialbe' is not defined`.

This makes it obvious that I mispelled the name and wrote `varialbe` instead of variable.

Another example of error occurs when you try to do invalid operations, such as adding an integer (Natural number) to, say, a string (of characters):

```
>>> s = 'a little string of size '
>>> 'This is ' + s + '.'
'This is a little string of size.'
>>> s + 26
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-1ffac3e9a4d8> in <module>()
----> 1 s + 26

TypeError: Can't convert 'int' object to str implicitly
```

1.2 Exercise: revision on type-casting

Make the previous example work by appropriately using explicit type-casting (which Python just told you cannot be done implicitly).

1.3 Let's play catch

In Python, you can **try** to run some code and **catch** any exception that might occur. This is done with the syntax:

```
try:
    'block of code'
except ExceptionName:
    'code in case of error'
```

Reusing our previous example, consider a simple function that sums two elements, let's call it `my_sum` to not override the built-in `sum`:

```
def my_sum(a, b):
    return a + b
```

As you previously witnessed, summing a string and and int will result in an error:

```
>>> my_sum('a', 2)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a17a53c00a15> in <module>()
----> 1 my_sum('a', 2)

<ipython-input-1-931f06bac813> in my_sum(a, b)
      1 def my_sum(a, b):
----> 2     return a + b

TypeError: Can't convert 'int' object to str implicitly
```

You can try to catch such errors and correct the code as you did in the previous exercise:

```
def my_sum(a, b):
    try:
        return a + b
    except TypeError:
        if type(a) is str and type(b) is int:
            return a + str(b)
        elif type(a) is int and type(b) is str:
            return str(a) + b
```

Now, if you try to sum a string and an int:

```
>>> my_sum('a', 2)
'a2'
>>> my_sum(2, 'a')
'2a'
```

And if you compose the function calls:

```
>>>my_sum(my_sum('R', 2), my_sum('D', 2))
'R2D2'
>>> my_sum('C', my_sum(3, my_sum('P', 'O')))
'C3PO'
```

The old ones have always been better...

2 A bit on Ipython

Ipython is a great interpreter and offers enhanced capabilities for Python programming. It supports command history. Use the `up` and `down` keys to navigate previously typed commands.

Now we are going to present some Ipython functions. These functions work **only in Ipython**, they **are not Python** code and you **should not** type them inside your script, but **only in the Ipython promt**.

2.1 Getting help

It is important that you are able to get help for example when you do not know how to use a function.

To get help on a function, in Ipython, you can use the `?` after the name of the function.

```
function?
```

Replace `function` by the name of the function you need help on.
Similarly, to *view the source* code of a function, type:

```
function??
```

If you are in a normal python interpreter, you can always type:

```
help(function)
```

2.1.1 Example

Let us define a list of integer named l:

```
l = [1, 2, 3, 4]
```

Calculate the sum of the elements of this function using the build-in function `sum`.
You can display information about this function as explained above.

```
>>> help(sum)
Help on built-in function sum in module builtins:

sum(iterable, start=0, /)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

Turns out you can just do `sum(l)` to get the sum of the elements of a list...

2.2 Using system command from Ipython

Lines starting with ! are passed directly to the system shell, and using !! or var = !cmd captures shell output into python variables for further use. For instance the following command saves `print("hello world!")` in a file called test.py.

2.2.1 If you are using linux/mac os:

You can create a file `test.py` as follows:

```
!echo 'print("hello world!")' > test.py
```

You can use any unix command: you can list the files in your current directory with `ls`:

```
!ls # list what is in the folder (you should see test.py now)
```

If you are unsure which folder you are in, you can print that using `pwd` (*print working directory*):

```
!pwd # print working directory, i.e. which folder you are in
```

2.2.2 If you are using windows:

You can create a file `test.py` as follows:

```
!Echo print("hello world!") > test.py
```

You can use all windows commands. For instance, to list the files in the current directory, you can use:

```
!dir # list files in the current directory
```

And if you are not sure which directory you are in:

```
!cd # current directory
```

2.3 Run a script and time a function from Ipython

To run a script called `test.py`from ipython, simply type:

```
%run test.py
```

%timeit allows you to time the execution of short snippets using the timeit module from the standard library:

```
%timeit my_function(arguments)
```

This will execute my_function several time and return the average runtime.

2.4 Debugging

If your program is crashing or not running correctly, the first thing in debugging the problem is usually using printing. You should inspect your code and print variables during the execution to make sure they have the correct values.

In case your code fails, Python comes with a debugger that allows you to run your code line by line just before the error occurs and figure out what went wrong. Think of the debugger as a small interpreter you can run just before you program crashes.

We'll not go into much detail in testing and debugging your code. Much more detail is given in your textbook's chapter 7.

2.4.1 Debugging in Spyder

Debugging is convenient in Spyder. See the debugging menu: you can start debugging your file using `Ctrl+F5`. You can then run the file line-by-line using `Ctrl+F10`. Inspect the blue buttons in the toolbar and the `Debug`-menu: these provide other debugging options. These allow not just line-by-line execution, but also stepping into a function to see how it executes. If you want to skip to a specific point of code to debug, you can also set a breakpoint on that line, and run the code until that breakpoint. Try the Spyder debugger on the following code.

```
a = 'a'
b = 2
a + b
```

2.4.2 Debugging from a Python script

Inside your Python script (or python code), you can enter the debugging at any line by typing the following:

```
import ipdb;
ipdb.set_trace()
```

This will start an interactive session *exactly at this point* and you will be able to inspect the value of the variables, and check where bugs could be comming from.

In Ipython, you can type `%pdb`to automatically call the debugger after a crash. Alternatively, `%debug` allows you to enter post-mortem debugging.

2.4.3 Inside the debugger: debugging commands

In the debugger you can use use the following commands:

- p var (for print) to print the value of the variable `var`
- in practice, you can use pp instead (for *pretty print*), it looks less horrible
- pp locals() to list variables in the current scope of your function
- pp globasl() to print the variables in the global environment
- l to *list* the code line around your call (list)
- n to go to the next line in your current script/function
- s to *step into* (eg if the next line a a function call, don't just execute that line, but also **step inside** that function)
- a prints the **arguments** of the current function call
- j line_number **jumps** to that line number
- u to go up the stack
- d to go down in the stack
- when you are done, use c to continue the execution of your code and exit the debugger.

2.4.4 Example using Ipython

After a crash in Ipython, as we said, you can enter the debugger after a program crashed using the magic `%debug`:

```
In  [1]: a = 'a'
         b = 2
         a + b
Out [1]: ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last
         <ipython-input-2-f8441d05e3b7> in <module>()
               1 a = 'a'
               2 b = 2
         ----> 3 a + b

         TypeError: Can't convert 'int' object to str implicitly
```

Your program crashed: in Ipython, you can start the debugger to enter the program **just before** the crash occured:

```
In  [2]: %debug
Out [2]: > <ipython-input-2-f8441d05e3b7>(3)<module>()
               1 a = 'a'
               2 b = 2
         ----> 3 a + b

           ipdb>   p a
         'a'
           ipdb>   p b
         2
           ipdb>   l
               1 a = 'a'
               2 b = 2
         ----> 3 a + b

           ipdb>
```

Note when debugging: Python **first** checks the syntax of the **while script** before it starts executing your code. So if you have a syntax error in the last line of your program, this will still make your program crash before the first line got executed!

3 Exercises

3.1 Exercise: divisions

Use the debugger to find the mistake (it is obvious this time):

```
>>> def divide(a, b):
...     """divides a with b"""
```

```
...     return a/b
>>> divide(1, 0)
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-9-18ba57c3b8d9> in <module>()
      3     return a/b
      4
----> 5 divide(1, 0)

<ipython-input-9-18ba57c3b8d9> in divide(a, b)
      1 def divide(a, b):
      2     """divides a with b"""
----> 3     return a/b
      4
      5 divide(1, 0)

ZeroDivisionError: division by zero
```

What happened?

Re-write the previous function using the introduced try/catch so that incorrect divisions returns "NaN" (not a number) instead of raising an exception.

3.2 Exercise: factorial

Consider the following function fact(n). Given an integer n, it should return $n!$.

```
def fact(n):
    """ A buggy factorial"""
    accumulator = 1 # fact(0) = 1
    for i in range(1, n):
        accumulator*= i
    return accumulator
```

However, it is incorrect. Find and correct the mistake.

4 Python: more built-in types

4.1 Tuples

Tuples are like list but they are not mutable:

```
>>> a = (1, 2, 3)
>>> a
(1, 2, 3)
>>> type(a)
tuple
>>> a[0]
1
>>> a[2]
```

```
3
>>> a[-1] # Last item
3
```

Since they are immutable, if you try to change a value, you will get an error:

```
>>> a[0] = 2
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-48f8c1dbf98d> in <module>()
----> 1 a[0] = 2

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

4.2 Dictionaries

Dictionaries are a very useful structure in Python. You can think of them as list that you access using a **key** instead of an integer index.

You can create a dictionary using the keyword `dict`. To add an element to a dictionary, the syntax is `dictionary[key] = value`. You can also directly create a filled dictionary `{key_1:value1, key_2, value_2, ..., key_n: value_n}`.

Under the hood, dictionaries are implemented as *hash table*, but we will not go into detail here. However, a consequence is that you can use any *hashable* object as a key while values can be any object.

```
>>> d = dict()
>>> d['key_1'] = 'value_1'
>>> d['key_2'] = 'value_2'
>>> d['key_3'] = 'value_3'
>>> d['key_4'] = 'value_4'
>>> d
{'key_1': 'value_1',
 'key_2': 'value_2',
 'key_3': 'value_3',
 'key_4': 'value_4'}
```

You can go through the elements of a dictionary:
You can also access the keys or the values

```
>>> list(d.keys())
['key_4', 'key_3', 'key_1', 'key_2']
>>> list(d.values())
['value_4', 'value_3', 'value_1', 'value_2']
```

Usually we iterate through the keys, the values, or both:

```
>>> for key in d:
...     print(key, d[key])
key_4 value_4
```

```
key_3 value_3
key_1 value_1
key_2 value_2

>>> for key, value in d.items():
...     print(key, value)
key_4 value_4
key_3 value_3
key_1 value_1
key_2 value_2

>>> for value in d.values():
...     print(value)
value_4
value_3
value_1
value_2
```

Notice that the elements of a dictionary are **NOT** ordered. If you want ordered dictionary, you can use the `OrderedDict` class from the standard library:

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d['key_1'] = 'value_1'
>>> d['key_2'] = 'value_2'
>>> d['key_3'] = 'value_3'
>>> d['key_4'] = 'value_4'

>>> for key, value in d.items():
...     print(key, value)
key_1 value_1
key_2 value_2
key_3 value_3
key_4 value_4
```

4.3 Sets

Sets allow you to create sets of unique elements and can be created using the keyword `set`:

```
>>> a = [0, 1, 1, 0, 2, 3]
>>> set(a)
{0, 1, 2, 3}
>>> set([0, 1, 1, 0, 2, 3, 'a', 'b', 'a'])
{0, 1, 2, 3, 'b', 'a'}
```

Alternatively you can use create a set like a list by replacing `[` and `]` by `{` and `}` respectively:

```
>>> a = {0, 1, 1, 0, 2, 3}
>>> type(a)
```

```
set
>>> a
{0, 1, 2, 3}
```

Basic operations on set: union (set composed of all elements of both sets) and intersection (set composed of only the common elements):

```
>>> a = {0, 1, 2, 3}
>>> b = {2, 3, 4, 5}
>>> a.union(b)
{0, 1, 2, 3, 4, 5}
>>> set.union(a, b) # Equivalent
{0, 1, 2, 3, 4, 5}
>>> a.intersection(b)
{2, 3}
```

4.4 Exercise: reserved keywords

Consider the following snippet of code:

```
>>> a = list([1, 2, 3, 4])
>>> sum(a)
10
>>> sum = [4, 5, 6]
>>> c = [1, 3, 5]
>>> sum(c)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-af6fce17fe35> in <module>()
----> 1 sum(c)

TypeError: 'list' object is not callable
```

What happened?

4.5 Operations on files

Download the file ebook.txt from the hub (it should be called 'ebook.txt') and save it in the **current folder**:

If you have a doubt what your current folder is execute the following snippet:

```
from pathlib import Path
Path('.').absolute()
```

This will print said current directory.

You can also list what is in said directory:

```
from pathlib import Path
list(Path('.').iterdir())
```

Or list the txt files in the directory: `python >>> from pathlib import Path >>> list(Path('.').glob('*.txt')) [PosixPath('book.txt'), PosixPath('iliad.txt'), PosixPath('moliere.txt'), PosixPath('odyssey.txt')]`

```
In [2]: filename = 'ebook.txt'
```

Opening a file is easy: just use open(filename, FLAGS), where flags can be 'r' if you are reading the file, 'w' if you are writing in it (they are more options, Google it if you are interested).

Let us read the file you just downloaded:

```
In [3]: f = open(filename, 'r')
```

If you have a doubt what your current folder is execute the following snippet:

```
from pathlib import Path
Path('.').absolute()
```

This will print said current directory.

You can also list what is in said directory:

```
from pathlib import Path
list(Path('.').iterdir())
```

Or list the txt files in the directory: `python >>> from pathlib import Path >>> list(Path('.').glob('*.txt')) [PosixPath('book.txt'), PosixPath('iliad.txt'), PosixPath('moliere.txt'), PosixPath('odyssey.txt')]`

Now you can read from this file:

```
In [4]: l = f.readline()
```

This reads **one** line:

```
In [5]: print(l)

        On an exceptionally hot evening early in July a young man came out of the g
```

If you use readline again it will read the next line

Or you can iterate through the lines. Note that in the example below, we set the maximum number of bytes to read to not have to go through the whole book.

```
In [6]: for l in f.readlines(120):
            print(l)


        He had successfully avoided meeting his landlady on the staircase. His garr
```

Any idea who the book might be from and what its title could be? :)

4.6 Open exercise:

Write a function to read a book and count the occurences of each word (i.e. their frequency) in the book.

Hint: you might want to get rid of the punctuation.

4.7 Bonus

Now you can try to download a book from the Gutenberg project and do the same kind of exercise! However you might encounter some problems with encoding...

```
In [7]: urls = {'odyssey': 'http://www.gutenberg.org/cache/epub/1728/pg1728.txt',
                'iliad': 'http://www.gutenberg.org/cache/epub/16452/pg16452.txt'}
```

If you wish you can use the following function to dowload them for you. If you have any problem simply dowload them manually, don't try to spend too much time on this. In any case **do NOT** execute this code too many times otherwise the website will ban your IP :) Trust me, it happened to me while writing this tutorial...

```
In [5]: import requests

        def save_from_url(url, filename):
            response = requests.get(url, stream=True)
            if response.status_code == 200:
                with open(filename, 'wb') as f:
                    for chunk in response:
                        f.write(chunk)

        for (name, url) in urls.items():
            save_from_url(url, name + '.txt')
```