Contents

Getting prepared

Tools

The tools used in this course (Spyder and the Jupyter Notebook server) can be found in the Anaconda distribution. Once you have installed this distribution (which should already be present on your company notebook), you can start these tools by:

- selecting the individual tool from the Anaconda3 menu in the Start menu,
- going to the Anaconda Navigator and use it to start them,
- selecting the Anaconda Prompt in the Anaconda3 menu.
  The last option opens a Command Line window with the proper environment setting. In there you can cd to the folder in which you have placed our notebooks and can start the Jupyter Notebook there by typing: jupyter notebook . If you are being asked for a token, check this Anaconda Prompt window for it.

Course notes

We will be using two notebooks:

For a quick overview of Python: https://github.com/jakevdp/WhirlwindTourOfPython

For an intro to the packages we will use (Numpy, Pandas, Matplotlib):

https://github.com/jakevdp/PythonDataScienceHandbook

Both of these are also available as pdf's:

http://www.oreilly.com/programming/free/a-whirlwind-tour-of-python.csp

https://jakevdp.github.io/PythonDataScienceHandbook

Using the tools

Even though the tools are basic and generally easy to use, it will take some time to get used to them. The first chapters (notebooks) of the DataScience Handbook provide some basic information. I would read those first and then take your time to explore both IPython and the notebooks themselves. You may also want to compare using plain Python versus Ipython.

Debugging

You can write your Python scripts in any text editor and then keep running and modifying them until they no longer produce errors and actually deliver the results you were aiming for. Syntactic errors are rarely hard to find, but semantic errors (the script runs, but gives unexpected results or crashes for certain inputs) are often much harder to isolate. The most basic tool at hand is a rich sprinkling of print statements at points of interest in your code.

A debugger is a tool that basically makes this an easier and interactive process: you get to put breakpoints in your code, which when reached will cause execution to halt and allow you the opportunity to print the current value of any name (variable) bound at that point. You can then step through the remaining code line by line, pausing to repeat these inspections, or you can continue the execution up to the next breakpoint, if present or else finish the script.

The Python standard library comes with a basic debugger that allows you to do just this: pdb

When you import pdb, you can place pdb.set_trace() statements where you want your code to "break". The pdb debugger will be started and show its prompt, waiting for further instructions. For an easier to read intro to pdd, look here.

IDE's such as Eclipse with Pydev or PyCharm have built very nice debugging interfaces on top of this basic debugger. If you are used to these tools, using the debugger in Spyder will be a let-down as it does little else besides giving you an alternative option to introduce breakpoints by clicking in the margin to the left of the line you want to stop at, and by giving you a few buttons as an alternative to typing the corresponding (i)pdb commands directly to its prompt in your IPython console.

Modules and packages

The slides can be found here, look here for some simple demos.

Some of the traps you may walk into using imports and setting up packages can be found here:

http://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html

If working with virtual environments and/or creating distributions has no secrets for you, you may want to take a look at these.

1. Create a virtual environment, activate and install some package using pip.
2. Create trivial script using package.
3. Create requirements listing.
4. Probably for a later time: make your own distribution, put it on the test PyPI server, install it and remove it again.

See https://hynek.me/articles/sharing-your-labor-of-love-pypi-quick-and-dirty/ for general instructions. See https://packaging.python.org/guides/using-testpypi/ for instructions on how to use the test server

Exercises

A number of exercises and their solutions can be found here.

Calculator

Build a simple calculator. See *calculator.py* for some suggestions.

Google exercises

The Google exercises are mainly about string manipulations / searches. They involve reading and writing files and passing arguments to your scripts.

Mutable data

The code below is an attempt at writing a function **add** that adds a value to a list of values stored under the k key in table. When the key is new, the function should give the caller the option to specify an initial list of values to use.

There are several things that go wrong here. Why? How would you resolve this?

```
table = {}
last_added = ()
def add(k, v, start=[]):
    last_added = (k,v)
    print("Adding: ", last_added)
    if k in table:
        table[k].append(v)
    else:
        table[k] = start.append(v)
add("x", 2)
add("y", 5)
add("x", 5)
```

```
add("z",3,[2])
add("z",4)
print("Last added to table {}: {}", table, last_added)
```

Build-your-own Python

One can learn a lot from simply trying to implement some of the built-in functions, such as map, filter and reduce.

With the knowledge on iterators in mind, you could also try to write your own for statement (as a function), using the **exec** function that allows you to execute any string as a Python block. This is just to understand how for works, you should never include exec in production code.

Comprehension

1: List Comprehension

```
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Write a list comprehension that takes this list and makes a new list with only the even elements in this list.

2: Another List Comprehension

Find all x where x is a natural number less than or equal to 100 and x is a perfect square. This can be solved using a for-loop as:

```
for i in range(1,101):     #the iterator
   if int(i**0.5)==i**0.5: #conditional filtering
     print i               #output-expression
```

Create a list comprehension doing the same

3: Matrix Flattening

Take a 2-D matrix as input and return a list with each row placed on after the other.

The Python code with a FOR-loop could look as follows:

```
def flatten(matrix):
    flat = []
    for row in matrix:
        for x in row:
            flat.append(x)
    return flat
```

Create a matrix (using list comprehension!) and test flatten. Create a list comprehension version that achieves the same result.

Other options to pursue:

- Extend to n-dimensional matrices
- Or to trees

4: Dictionary Comprehension

Take two lists of the same length as input and return a dictionary with the first list as keys and the second as values.

The Python code with a FOR-loop could look like this:

```
def makedict(keys, values):
    dic = {}
    for i in range(len(keys)):
        dic[keys[i]] = values[i]
    return dic
```

Create a solution using dictionary comprehension, using e.g.:

country = ['Germany', 'France', 'Belgium', 'England', 'Spain', 'Italy']

capital = ['Berlin', 'Paris', 'Brussels', 'London', 'Madrid', 'Rome']

Hint: look at the `zip` generator.

Dictionary comprehension is just one way to achieve this. Look up the dict type and see what other (built-in) options there are.

Generator

1. *infinite number generator*
   Write function that generates infinite number of integers.

2. *firstn*
   Write a generator that generates first n items of an iterator.

3. *Fibonacci*
   Write Fibonacci function as generator