Advanced Quantum Mechanics & Spectroscopy (CHEM 7520)

Alexander Yu. Sokolov

Introduction to Programming with Python

Python is a very powerful high-level programming language that can be used to automate tasks and implement complicated computer algorithms in just a few lines of code. Thanks to its extensive mathematics library and the third-party libraries, such as Numpy and Scipy, Python finds many applications in scientific data analysis and computer simulations. This short overview is only intended to illustrate some of the Python's capabilities and is by no means comprehensive. For more details, please refer to the information available on the Internet.

A. Setting up Python

The latest versions of Python can be downloaded from the official Python website (`http://www.python.org/downloads/`). Note that Python versions are usually denoted as `x.y.z`, where `x`, `y`, and `z` are three numbers that specify a specific version (e.g., `2.7.14`). There are two widely used versions of Python: 2 (`x` = 2) and 3 (`x` = 3). While most of the Python features are supported by Python 2 and 3, some features are only supported by either of these two versions. Although Python 2 is still being used, it is highly recommended to use Python 3. Since January 2020, Python 2 is no longer supported by the Python developers.

Specific instructions on how to install Python on your computer depend on the type of the operating system and can be found online. For Windows, it is usually easier to download the installer from the official Python website and install the program to `C:\`. Most of the Mac operating systems already have some version of Python installed. To check that, open Terminal and type `which python`. This should return the path to the Python executable. If the executable is located, run `python --version` to find out which Python version is installed. If you don't have Python installed, you can either download it from `www.python.org` or install it using the Brew packaging manager (`http://brew.sh`).

B. Setting up Numpy

While Python is very powerful and has a lot of functionality by itself, there are many libraries that make it even more powerful. One of the most useful libraries for scientific research is Numpy (`www.numpy.org`). In addition to supporting many useful mathematical functions, Numpy allows to work with scientific data very efficiently by storing it in the form of numpy arrays. For more details on Numpy functionality, see sections below.

Since most of the Python distributions do not have Numpy installed by default, some basic instructions are provided here. For more information, please go to `www.numpy.org`. To check if Numpy is installed on your computer, open the Python interpreter (run the Python executable) and in the command line type `import numpy`, e.g.:

```
[Silverblade:~] alex% python
Python 2.7.13 (default, Mar 23 2017, 10:12:46)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

If running the `import` command does not result in any errors, your Python should be configured to work with Numpy. If you get an error, then Numpy needs to be installed on your computer. Follow the instructions below:

- **Windows:** Go to the directory where you installed Python. Inside of that directory, open the directory called `Scripts`. Inside of it, create a text file called "local.bat". Edit this file using Notepad and type `cmd` inside of it. Right click on the file and select "Run as administrator". This will open a command prompt window. Inside of the command prompt, type `pip install numpy`. This will download and install Numpy on your computer.

- **Mac OS X:** If you have Python installed on your Mac, first thing you need to do is to check if you have `pip` available. Open Terminal and type `which pip`, this should locate the path to `pip` if it is available. If you don't have `pip` installed, run `easy_install pip`. Then, open a new Terminal tab and run `pip install numpy`. Test if Numpy is installed by executing Python and running `import numpy`.

C. Python interpreter vs Python scripts

There are two ways to work with Python. First is to run Python as an executable. To do this on Windows, run `python.exe` from the installation directory. On a Mac, open Terminal and run `python`. You will see something like this:

```
Python 2.7.10 (default, Jul 15 2017, 17:16:57)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This way of using Python is called Python interpreter. Here, you can type Python commands and the interpreter will execute them line by line. For example:

```
>>> print("Hello world")
Hello world
>>> x = 4.0 * 12.0
>>> print(x)
48.0
```

While this can be very convenient for testing some of the Python commands, using Python interpreter for writing long programs that involve many lines can be difficult. To exit the Python interpreter type `exit()`.

Another way to use Python is to write Python commands in a separate file (so-called script) and execute them all at once later. To do that, create a file with an extension `.py`, write your program inside of this file and save it. On Windows, this can be done using Notepad. To create a Python-readable file on a Mac, open TextEdit, click on the Format menu and choose "Make plain text" option, then write your code and save. Alternatively, if you downloaded Python from `www.python.org`, you can create and edit Python scripts using the program called IDLE that is included in the Python installation (available in both Windows and Mac distributions). To run the Python program on a Mac, open the Terminal, locate the Python file and type `python my_program.py`. On Windows, you can do the same from the command prompt if your program is located in the same directory where `python.exe` was installed. To run your program from any directory, the path to your Python installation directory needs to be included in the Windows environment variable called `Path`. You can find instructions on how to change the Windows `Path` variable on the Internet.

D. Python object types

As in any programming language, data used in Python can be represented in different ways. There are two general types of data (or, generally speaking, objects): numbers and container objects. Here's a very brief description of them:

- **Numbers**

  1. *Integers*

     ```
     >>> x = 100
     >>> type(x)
     <type 'int'>
     ```

     Note that in Python 2, division of an integer by an integer will result in an integer:

     ```
     >>> y = 200
     >>> x/y
     0
     ```

     This is generally not the case in Python 3.

  2. *Real floating-point numbers*

     ```
     >>> y = 200.0
     >>> type(y)
     <type 'float'>
     ```

  3. *Boolean numbers*
     These are used in logical operations to specify whether a statement is true or false:

     ```
     >>> conv = True
     >>> type(conv)
     <type 'bool'>
     >>> conv = True and False
     >>> conv
     False
     >>> conv = True or False
     >>> conv
     True
     ```

  4. *Complex numbers*

```
>>> x = 1.0 + 1j * 2.0
>>> x
(1+2j)
>>> x**2
(-3+4j)
```

- **Container objects**

  1. *Lists*
     Lists are collections of objects denoted by square brackets `[` and `]`. They can be used to store different types of objects. New elements can be added to the existing lists, existing elements can be removed from them. Lists can be combined (concatenated) and also can be accessed by a range of elements (so-called slicing). Some basic examples are shown below:

     ```
     >>> x = [2.0, 3.0]
     >>> x.append(4.0)
     >>> x
     [2.0, 3.0, 4.0]
     >>> y = [5, "string"]
     >>> z = x + y
     >>> z
     [2.0, 3.0, 4.0, 5, 'string']
     >>> z[1]
     3.0
     >>> z[1:3]
     [3.0, 4.0]
     >>> len(z)
     5
     ```

  2. *Tuples*
     Tuples are also collections of objects, they are denoted with parenthesis `(` and `)`. Contrary to lists, once created, tuples cannot be modified:

     ```
     >>> a = (1,2,3)
     >>> a
     (1, 2, 3)
     >>> a[0] = 4
     Traceback (most recent call last):
     ```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Other than that, their functionality is similar to that of lists:

```
>>> print(a[0])
1
>>> print(a[0:2])
(1, 2)
>>> z = (4,5,6)
>>> a+z
(1, 2, 3, 4, 5, 6)
>>> len(a+z)
6
```

3. *Strings*
   Strings are used to represent symbolic data, they are denoted using single or double quotes. Similarly to lists and tuples, they can be accessed by elements or range of elements. Strings can be concatenated, but cannot be modified by an element assignment.

```
>>> str = "Hello world"
>>> len(str)
11
>>> str+" "+str
'Hello world Hello world'
>>> str[:6]
'Hello '
>>> str[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

   In addition, strings can be used to represent data in a particular format:

```
>>> str = "%10.5f %4d %4d" % (12345.678901234, 10, 20)
>>> print(str)
12345.67890   10   20
```

4. *Dictionaries*
   Dictionaries can be thought of "associative arrays", they are denoted

using curly brackets { and }. Unlike lists or tuples that, which elements are indexed by a range of numbers, elements of dictionaries are indexed by keys. Keys can be numbers, strings or even certain types of tuples. The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair. Some examples of dictionaries and their operations:

```
>>> dict = {}
>>> dict['apples'] = 5
>>> dict['oranges'] = 10
>>> dict['plums'] = 7
>>> print(dict)
{'plums': 7, 'apples': 5, 'oranges': 10}
>>> dict.keys()
['plums', 'apples', 'oranges']
>>> del dict['plums']
>>> print(dict)
{'apples': 5, 'oranges': 10}
>>> new_dict = {'carrots': 10, 'potatoes': 2, 'onions': 3}
>>> print(new_dict)
{'potatoes': 2, 'carrots': 10, 'onions': 3}
```

Note that the `type()` function can be very useful to determine the type of the Python object:

```
>>> type("Hello world")
<type 'str'>
>>> type([12.0])
<type 'list'>
```

E. Looping and condition statements

We use an `if` condition statement when a part of the code needs to be run only if condition is true. To denote a part of the code that belongs to an `if` statement, Python uses indentation. While the content of an `if` statement can be indented by any number of backspaces, it is recommended to consistently use 4 spaces:

```
>>> names = ['Mary', 'Alex', 'Susan']
>>> if 'Jack' in names:
```

```
...     print("I found Jack")
... else:
...     print("I didn't find Jack")
...
I didn't find Jack
```

The looping statements are used when a part of the code needs to be executed several times. The most widely used looping statements are `for` and `while`. The `for` statement is usually used when a part of the code needs to be run a known (predefined) number of times:

```
>>> for i in range(4):
...     print("This is my iteration #", (i+1))
...
This is my iteration # 1
This is my iteration # 2
This is my iteration # 3
This is my iteration # 4
```

In addition to looping over a range of numbers, Python's `for` statement can also loop over the elements of any predefined list:

```
>>> my_list = ['carrots', 'potatoes', 'onions']
>>> for veggie in my_list:
...     print("I need to buy", veggie)
...
I need to buy carrots
I need to buy potatoes
I need to buy onions
```

The `while` condition statement is used if we would like to repeat a part of the code for an undefined number of times until some condition is satisfied:

```
>>> x = 0.0
>>> while x < 0.8:
...     # Generate a random number multiple times
...     # until it is bigger than 0.8
...     x = random.random()
...     print(x)
...
```

```
0.753093820837
0.0225494984689
0.709880350986
0.807098413143
```

Note that the content of the `for` and `while` statements also requires indentation.

F. Matrix and tensor operations using Numpy

To be able to use Numpy for matrix and tensor operations, make sure that it is installed on your computer (see instructions above). Since Numpy is an external library, it is not loaded by default. Thus, every Python script that uses Numpy should contain the `import numpy` line. As any Python library, Numpy can be loaded with an alias:

```
>>> import numpy as np
```

The above line allows to shorten the Numpy function calls, e.g. `np.sqrt(4.0)` instead of `numpy.sqrt(4.0)`.

Numpy allows to efficiently store and manipulate data using the so-called *numpy arrays*. For example, a matrix can be represented using a two-dimensional numpy array as follows:

```
>>> A = np.random.random((4,4))
>>> print(A)
[[ 0.41876825  0.46548118  0.72390022  0.82810599]
 [ 0.91278577  0.6193322   0.89722354  0.92845717]
 [ 0.63918077  0.56033718  0.24796446  0.74611299]
 [ 0.07634179  0.81549647  0.95896538  0.01995572]]
```

In the above example, we used the Numpy function `random` of the `np.random` module to generate a 4×4 matrix with random floating-point numbers. To generate a zero matrix with the same dimensions, we can use the `np.zeros` function:

```
>>> W = np.zeros((4,4))
>>> print(W)
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

A powerful feature of the numpy arrays is that they can be used to represent tensors with more than two dimensions (up to 32). As an example, a $2 \times 2 \times 2 \times 2$ four-dimensional array can be created as:

```
>>> X = np.zeros((2,2,2,2))
>>> print(X)
[[[[ 0.  0.]
   [ 0.  0.]]

  [[ 0.  0.]
   [ 0.  0.]]]


 [[[ 0.  0.]
   [ 0.  0.]]

  [[ 0.  0.]
   [ 0.  0.]]]]
```

In this example, we created a numpy array that is filled with floating-point zeros. Note that in both examples above the dimensions of the numpy arrays are represented as Python *tuples*, e.g. `(4,4)` or `(2,2,2,2)`. Thus, the number of elements in a tuple defines the number of dimensions in a numpy array. There are many other ways to create numpy arrays. For example, a numpy array can be obtained by "stacking" two existing numpy arrays (using functions `np.vstack()` or `np.hstack()`) or by creating an empty array (using `np.array([])`) and appending data (using `np.append()`). Other Python data structures, such as tuples or lists, can be converted to numpy arrays as well.

Now let's go back to our first example and define a new matrix with dimensions 4×2:

```
>>> B = np.random.random((4,2))
>>> B
array([[ 0.19960546,  0.55020058],
       [ 0.97162975,  0.53432274],
       [ 0.17318349,  0.87559476],
       [ 0.94732401,  0.11567287]])
```

We can multiply the matrices **A** and **B** to compute a new matrix **C**:

```
>>> C = np.dot(A,B)
>>> C
array([[ 1.44571604,  1.20875634],
       [ 1.81889269,  1.72614007],
       [ 1.42177835,  0.95449993],
       [ 0.99258038,  1.319715  ]])
```

Note that the operation `A * B` is not a matrix multiplication, it is an element-wise multiplication instead. Since **A** and **B** are matrices of different dimensions, the operation `A * B` will result in an error. Multiplying `A * A` will result in a matrix **A** with all elements squared:

```
>>> A * A
array([[  1.75366845e-01,   2.16672731e-01,   5.24031531e-01,
          6.85759525e-01],
       [  8.33177858e-01,   3.83572377e-01,   8.05010082e-01,
          8.62032724e-01],
       [  4.08552054e-01,   3.13977755e-01,   6.14863745e-02,
          5.56684597e-01],
       [  5.82806936e-03,   6.65034499e-01,   9.19614603e-01,
          3.98230596e-04]])
```

Multiplication of several matrices can be performed in one line, e.g. for $\mathbf{D} = \mathbf{B}^{\mathbf{T}}\mathbf{AB}$ we can write:

```
>>> D = np.dot(B.T, np.dot(A, B))
>>> D
array([[ 3.24238683,  3.33394475],
       [ 3.12702581,  2.57578468]])
```

A more elegant way to perform such a matrix multiplication is to use the `reduce` function:

```
>>> D = reduce(np.dot, (B.T, A, B))
>>> D
array([[ 3.24238683,  3.33394475],
       [ 3.12702581,  2.57578468]])
```

where we used the `.T` attribute of the numpy array `B` to generate its transpose. An alternative way to compute the transpose of a matrix is using the `np.transpose()` function. *Note: To use the* `reduce` *function in Python 3, include the following*

line in your script: `from functools import reduce`.

So far, we have performed operations that involve all elements of a numpy array (e.g., matrix multiplication multiplies all elements of two matrices). To access a single element of an array, we can simply specify the indices of the element in square brackets (e.g., `A[3,2]` or `X[1,0,1,0]`, note that the 0-counting convention is used). Another very powerful way to access the subsets of elements of a numpy array is to use *slicing*. As an example, we can use slicing to print the third row of the matrix **A**:

```
>>> A[2,:]
array([ 0.63918077,  0.56033718,  0.24796446,  0.74611299])
```

Here, the first index 2 is used to specify the row, while the entry `:` specifies that all of the columns need to be accessed. Instead of accessing all columns, we can request to access only a subset. For example, we can restrict the range of columns to the second and third columns only:

```
>>> A[2,1:3]
array([ 0.56033718,  0.24796446])
```

Slicing can be used to set elements:

```
>>> A[2,:] = 1.0
>>> A
array([[ 0.41876825,  0.46548118,  0.72390022,  0.82810599],
       [ 0.91278577,  0.6193322 ,  0.89722354,  0.92845717],
       [ 1.        ,  1.        ,  1.        ,  1.        ],
       [ 0.07634179,  0.81549647,  0.95896538,  0.01995572]])
```

Alternatively, it can be used to access certain patterns of elements, such as elements with all even indices

```
>>> A[::2,::2]
array([[ 0.41876825,  0.72390022],
       [ 0.63918077,  0.24796446]])
```

or all odd indices

```
>>> A[1::2,1::2]
array([[ 0.6193322 ,  0.92845717],
       [ 0.81549647,  0.01995572]])
```

Slicing is not only very powerful, but can also be very computationally efficient. For other ways to access elements of the numpy arrays, please refer to the Numpy manual.

Finally, we consider how we can manipulate data stored in multi-dimensional numpy arrays. Much of the Numpy functionality available for the two-dimensional arrays can be used for the arrays with more dimensions. One difference is that, since multi-dimensional tensors have more than two indices, the multiplication of two tensors is no longer uniquely defined. For this reason, we will not use the `np.dot()` function to compute the product of two multi-dimensional arrays, but instead consider a more general function called `np.einsum()`. [Note that in certain cases the `np.dot()` function can be used to compute the product of two multi-dimensional arrays, see the Numpy manual]. To illustrate how `np.einsum()` works, let's consider the following tensor operation (so-called *tensor contraction*) as an example:

$$G_{xy} = \sum_{pq} E_{pq} F_{pxqy} \ , \tag{1}$$

where we multiply elements of the two-dimensional tensor (matrix) $\mathbf{E}$ by elements of the four-dimensional tensor $\mathbf{F}$ while summing over the two common indices to obtain a new two-dimensional tensor (matrix $\mathbf{G}$) as a result. Once $\mathbf{E}$ and $\mathbf{F}$ are computed and are stored as numpy arrays, the tensor contraction in Eq. (1) can be implemented as:

```
>>> G = np.einsum('pq,pxqy->xy', E, F)
```

The first argument of the `np.einsum()` function specifies how the tensor operation should be performed: the two indices of the tensor `E` and four indices of the tensor `F` are separated by a comma, while the indices of the third tensor `G` are separated by the `->` symbol, and the summation is performed over the repeated indices (so-called *Einstein convention*). The `np.einsum()` function is very flexible, it can perform operations with tensors of arbitrary dimensions (including the one- and two-dimensional arrays). For example, the norm of the tensor $\mathbf{F}$ ($N = \sqrt{\sum_{pqrs} F_{pqrs}^2}$) can be computed as:

```
>>> N = np.sqrt(np.einsum('pqrs,pqrs', F, F))
```

While `np.einsum()` can be used to perform complicated tensor operations in a compact form, it is less computationally efficient than `np.dot()`. For this reason,

using `np.einsum()` is recommended for the initial (pilot) implementations only, while for the efficient implementations it should be avoided. In the latter case, it is often possible to reduce the dimensionality of the multi-dimensional tensors by transposing the indices (using `np.transpose()`) and reshaping them down to the two-dimensional arrays (`np.reshape()`), which can be multiplied as "super-matrices" using the `np.dot()` function.

G. Defining new functions in Python

Programming computer algorithms often involves performing the same task multiple times. Such tasks can be conveniently defined as functions, which can be called in several places of the program. Python makes defining new functions very easy. We will illustrate this on a simple example. Let us define a function that takes a numpy array as an input and returns a list of its three largest elements. First, we define the function as:

```
>>> import numpy as np
>>> def max_elements(A):
...     # Turn a numpy array into a one-dimensional array
...     A = A.ravel()
...     # Sort the array in the descending order
...     A[::-1].sort()
...     # Return first three elements as a Python list
...     return A[:3].tolist()
...
```

We can now use this function to print the three largest elements of a random matrix:

```
>>> X = np.random.random((4,4))
>>> print(max_elements(X))
[0.9840955434199267, 0.952108251869013, 0.9236377591745225]
>>> X = np.random.random((4,4))
>>> print(max_elements(X))
[0.893858566081185, 0.7728361321722338, 0.746132464101978]
>>> X = np.random.random((4,4))
>>> print(max_elements(X))
[0.9711829325900079, 0.9198351757368676, 0.8354676131068733]
```

The function we defined above can also be used for arrays with more than two dimensions, e.g.:

```
>>> Y = np.random.random((4,4,4,4))
>>> print(max_elements(Y))
[0.990771191307948, 0.98952424412145, 0.982675647370982]
```

Although this is a very simple example, Python can be used to define more complicated objects, such as a function with a varying number of arguments or a recursive function that calls itself multiple times.

H. References

For more information, please refer to resources available online. Some of them are listed below:

- Python manual: http://docs.python.org/3/
- Numpy manual: http://docs.scipy.org/doc/
- Python Codecademy: https://www.codecademy.com/learn/learn-python-3
- Google's Python class: http://developers.google.com/edu/python
- Python tutorial: http://vergil.chemistry.gatech.edu/courses/python/index.html
- There are several good Python and Numpy tutorials available on YouTube