Data Structures and Algorithms with Python
Tutorial 1

1 Introduction to Python: getting started

The goal of this tutorial is to familiarise you with the Python programming language. Loosely speaking, a programming language is simply a way to give instructions to a computer, for example performing calculations on data and printing the results. Python is a free, widely-used language that can be used in a variety of application domains, and allows us to start writing code in a quick and friendly manner. It derives its name from the British comedy group Monty Python (not from the snake!)

The main part of the tutorial consists of four main sections:

1. Installing Python
2. Getting started with Python
3. Basic Python concepts
4. Building programs and functions

The objective is to get you started with Python with brief descriptions of relevant concepts. This part covers much of the material in chapters 2 through 4.1 in the Guttag book.

A set of short exercises follows that allows you to test your understanding of the concepts. You will start the exercises in this tutorial and continue them as preparation for discussion in the first lecture.

1.1 Installing Python

We will be using the Anaconda distribution of Python (probably named after a snake), which can be downloaded and installed for free. It also has the advantage of coming with all the Python packages you will need for this course.

If you have not done so yet, head to http://continuum.io/downloads and download the version corresponding to your operating system, for Python 3 (the current version should be 3.5). Anaconda is available for Windows, OS X and Linux.

Simply run the downloaded executable and follow the instructions.

2 Getting started with Python

2.1 Before you begin

The goal of the tutorial is to get you started with coding in Python. The best way to learn coding is to get your hands dirty with code, so try to type and execute as many of the commands as possible. Play around and try different things. And whenever you get stuck, ask for help!

2.2 Python as a programming language

Python is a general-purpose, modern, *high level* language. Loosely speaking, computers are able to only execute *low level* commands that are written in "machine code". High-level languages like Python therefore need to be processed into low-level code before they can be run. While this results in a speed disadvantage, it allows Python code to be much closer to human language than a lower-level language, say C++, or machine code like assembly. The big advantage is that Python programs are short and easy to read and write, as we shall see shortly.

Python is an *interpreted* language, meaning that the code is evaluated by an *interpreter* at run time. An interpreter essentially reads our Python programs line-by-line and executes the commands.

Next, we'll look at the interpreter in action. To do this, open the command line on your machine.

2.2.1 Opening the command line

Windows Press the Windows key and type cmd. This should open the start menu and find the Command Prompt.

OS X Use the Key combination Cmd-Space, type "terminal" and press Enter to open a Command Prompt

Linux You'll probably know this if you're using Linux... (Ctrl-T should work in most cases).

2.3 The Python interpreter

On the command line, start the python interpreter by typing `python` (the directory you're in does not matter).

You should be greeted by something that looks like this:

```
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.190(
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Check that you can correctly open the Python interpreter. Opening the interpreter prints out basic information, such as the version numbers of Python and Anaconda.

The >>> indicates the Python prompt, which you use to communicate with Python. This is the Python interpreter: Python is now waiting for you to type something to interpret it.

Let's type in a command:

```
print('Hello, world!')
```

The result should look like this:

```
>>> print('Hello, world!')
Hello, world!
>>>
```

This means that Python has interpreted you command (resulting in the display of the string 'Hello, world!') and has displayed a new prompt for you to type a command.

Let's try another command and see what happens:

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

This should of course print out 4.

Now that we have Python running, we're ready to go through some basic concepts.

3 Basic Python concepts

3.1 Objects, values and expressions

Objects are the key things that our Python programs manipulate. When we write

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

we get a result, 4, that is an object that has a *value* (here 4) and a *type*. To get an object's type, we can write:

```
>>> type(4)
<class 'int'>
```

The type is of course an *integer*. Similarly,

```
>>> type(4.0)
<class 'float'>
```

This is a *floating-point number* that is used to represent real numbers.

You've also witnessed here the `dynamically typed` nature of Python. In other words, you do not need to declare the type of the objects you introduce, as you would have to do in other languages like C. The Python interpreter infers the type of the variable from the context and you can access it using the `type` function.

What about "4.0"?

```
>>> type("4.0")
<class 'str'>
```

With the quotations, 4.0 becomes a *string*. A string can be defined in Python using either single or double quotes.

If an object turns out in an undesired format, you can convert it easily from one type to another:

```
>>> type(float("4.0"))
<class 'float'>
```

But what happens when you do the following?

```
int(4.4)
```

We can combine objects with *operators* to form *expressions*. We have already seen this in 2+2. For integers and floats, we can use arithmetic operations:

```
2 - 2
2.3*5.4
6/3
6//4
3**0.5
89%2
```

Try to figure out what these different operators do by running the code.

When using multiple operations including parentheses and such, the order of operations follows standard rules of arithmetic.

```
(2-4)**3*5
```

With these operations, we are essentially ready to replace our pocket calculators with Python. But building on them, we can go much further...

3.2 Variables and assignment

Often we want to keep track of objects, such as the results of our calculations. A *variable* is an object that is stored in working memory and accessed through an identifier. For example, here, we create a variable called `x` that contains the value `12`:

```
x = 12
```

Here we store the value `12` in memory and access it with the identifier `x`. In other words, `x` points to the variable `12` in memory. `x` is essentially a name for the underlying object.

You can display the value of any variable using the function print on its identifier:

```
>>> print(x)
12
```

To make our code easy to understand, we prefer variable names that are meaningful:

```
>>> pi = 3.14159
>>> radius = 5
>>> area = pi*radius**2
```

We can use arbitrarily long variable names, but there are some rules. What happens when you try the following?

```
>>> 1stradius = 5
>>> import = 8
>>> x* = 4
```

We have to start variable names with a letter, and we can not use certain keywords reserved for Python (see the textbook for these) and certain symbols.

3.3 User input

We often wish to interact with the program user to get input. The syntax for doing this is very intuitive in Python. Try out the following code

```
>>> name = input('Hello there! What is your name?')
>>> print('Your name is', name)
```

Here we're using Python's built-in `input` function to prompt the user for a name. We're then assigning the input to the `name` variable. As you can see, the `print` statement can take multiple inputs.

3.3.1 Exercise:

Write a simple code to ask the users for his/her age and store it in a variable. What is the type of that variable? Is it what you expected?

3.4 Comments in Python

Before we move on to the next section, let's look at how to write comments in your code. Comments are designed to help the reader understand your code and are ignored altogether by the interpreter. In Python, they are delimited by the character `#`.

When you see `#` in a python code, everything following that symbol, in the same line, is a comment and will be ignored by the interpreter:

```
x = 2 # This is a comment and will be ignored by the interpreter

# This is another comment
print(x)
# One last comment
```

With these ideas, we're ready to start building some more complex programs.

4 Building Python programs

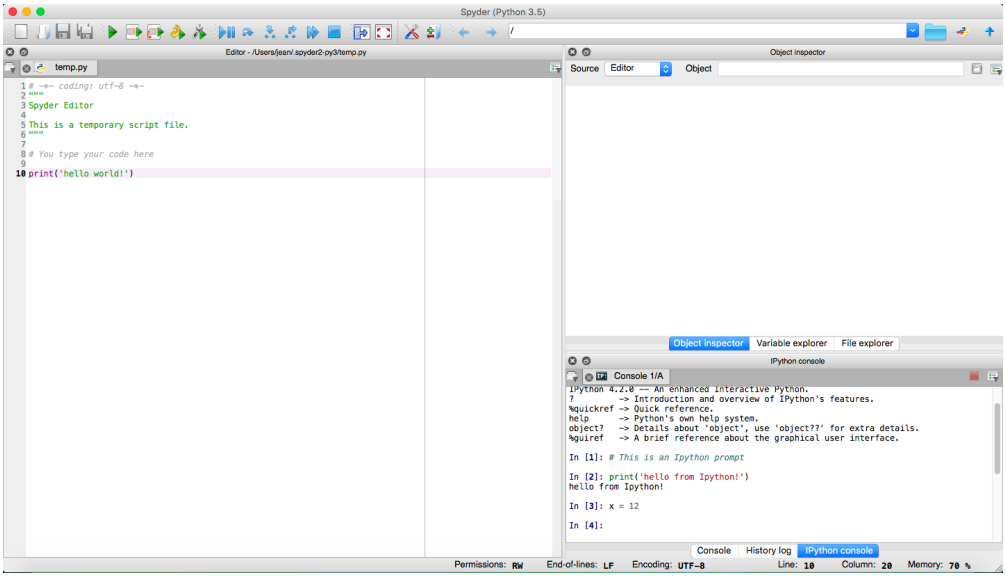
So far our code has been short snippets that are convenient to run in the Python interpreter. When our programs get longer this becomes quite inconvenient. Another way to run code is to write our program into a file and send it to the interpreter. These files are often called *scripts*: in Python, their extension is `.py`.

Scripts are a convenient way to save our code for future use. We can write them using any text editor, but many programmers use editors that are part of an integrated development environment (IDE). IDEs facilitate coding in several ways: our Anaconda installation comes with a widely-used one called Spyder.

4.1 Spyder IDE

Having installed Anaconda, you should find Spyder on your system. (In Windows, press the Windows key and start typing `spyder`.)

Go ahead and open Spyder: you should obtain an interface like the one depicted here:



The Spyder IDE

First look at the bottom right panel. This is an Ipython console, which is an enhanced version of the Python shell we were just using on the command line. The main visible difference is that instead of >>> it prompts you with `In [1]`. You can run all the above commands here as before.

The top right panel displays helpful information on your variables and functions (do not worry about this too much for now).

The left panel is a text editor, where you can type code. But different from the console, this code will not run immediately. Instead, you'll have to invoke Spyder's commands to run your code or parts of it.

When you create a new file with Spyder, it comes with a prefix, for instance:

```
# -*- coding: utf-8 -*-
"""
Created on Wed Aug 17 15:06:38 2016

"""
```

This is a comment area that allows you to write out what the script is doing (in English). Python will not run lines starting with # or anything between triple quotations """.

Let's start by creating a simple script. Write out the following code in the editor.

```
s = 'Hello world'
print(s)
```

To save your code, click on `File->Save As`, and find a suitable directory. You may want to create a specific directory for all the code from this module. Give your script a filename, for instance `helloWorld.py`, and save it in the directory. If you have code saved in multiple directories, you can navigate between them using the buttons on the top right corner of Spyder.

You can execute code in several ways in Spyder. First, to execute everything in the file open in the editor, you can select `Run->Run`, or simply press the green play button or hit the `F5`-key.

Go ahead and run the program. Notice what happens in the console window on the bottom right: this shows that you ran the file, and the printed output of the program.

Often you do not want to run the entire script, but perhaps just a few lines. To do this, select the code you wish to run, and select `Run->Selection or Current Line`, or hit the `F9`-key. Try this out and see what happens in the console.

Note that you can have many different functions and code snippets in a single `.py`-script file. You can, for example, use a single file to try out the examples and exercises below. Running only a few lines of selected code through the `F9` key then becomes very useful.

Spyder is not the only way to run your Python scripts. A classic way would be to go back to the command line where we started at and navigate to the directory where we saved `helloWorld.py`. You could then type `python helloWorld.py` to execute our code (the entire file). However, an IDE like Spyder allows you to conveniently write longer programs whilst also being able to quickly test parts of your code in the console.

You will be using Spyder (or another IDE if you prefer) throughout the module to work on assignments.

4.2 Booleans and controlling program flow

Our programs so far have been *straight-line programs*, where we execute one statement after another in a specified order. But often we only want to execute code based on some conditions. For example, if we're writing a program to calculate a square root of a number, we may only want to run our code if the number is non-negative.

For this, we need Boolean variables.

4.2.1 Booleans and comparisons

In Python, a Boolean `bool` is a type of object like float or int. It represents a binary variable taking values `True` or `False`.

For example, we may want to compare if two variables are equal or different. Recall that we used the = operator for assigning a value:

```
x = 50
```

To compare whether variables are equal, we need to use the double equals sign == instead. Run the following code in Spyder.

```
# Assigning values
a = 10
b = 10
c = 11
# Comparing values
a == b
a == c
a = c # Another assignment
print(a)
```

Here we're first assigning integer values to `a,b,c`. We then compare whether these are equal, resulting in Boolean output (you can check this using `type(a==b)`). The last line is an assignment again. For comparisons, we can also use

```
# Comparing assigned values
a != b Difference
a >= c
a < b
# Comparing the comparisons...
a > b and a == c
a == c or a == b
not a > b
```

Try these expressions out in Spyder. What happens? The first set of expressions use comparison operators to compare the variables `a,b,c`, resulting in Boolean expressions. The second set of expressions are comparisons between the different Boolean expressions using operators `and,or,not`, resulting in new Boolean expressions.

4.3 Conditionals

Now we're ready to depart from straight-line programs. Suppose we want to write a program that prints out whether a number `x` is positive or negative.

We can do this in Python using a conditional `if` statement. The statement works as follows:

```
if CONDITION:
    # Block of code
else:
    # Another block of code
```

The `CONDITION` here should be a Boolean expression. If the Boolean is True, the first block of code is executed, if not, the `else` code block is executed. That is, our program would be:

```
if x >=0:
    print('Positive')
else:
    print('Negative')
```

Notice the colon `:` required to initiate the conditionals. Furthermore, note the indentation of the code. Python requires appropriate indentation (4 spaces) to recognize the code as part of the conditional block.

Try out the following code in Spyder.

```
temp = 38
humidity = 90

if temp >=30:
    print('Too hot!')
elif temp <= 0:
    print('Too cold!')
else:
    print('Perfect!')

print('Done with testing!')
```

This is an example of using `elif`, short for *else if*. This allows several cases for a condition. Try adding another one that prints out `Too moderate!` if `15 <= temp <= 25`.

Conditional statements can also be nested:

```
if CONDITION:
    # block of code
    if ANOTHER_CONDITION:
        # another block of code
```

Try replacing the code block after else with the following code. Be careful with indentation: it also has to be nested.

```
if humidity < 80:
    print('Perfect!')
else:
    print('Too humid!)
```

4.3.1 Exercise on conditionals

Write a small program that asks the user for his/her age. If the age is higher than 18, then print 'you can drink', otherwise, print 'sorry you cannot yet drink' and print the remaining number of years before they can drink.

Solution

```
In  [1]:age = input('How old are you?')
        # Remember, the result of an input is a str, we need to "type-cast" it (cha
        age = int(age)

        # Test if age is higher than 18
        if age > 18:
            print('you can drink!')
        else:
            print('Sorry you cannot drink yet.')
            remaining_years = 18 - age
            print('Still {} years before you can.'.format(remaining_years))

Out [1]:How old are you?17
        Sorry you cannot drink yet.
        You have to wait 1 years before you can.
```

4.4 Functions

Often we want to evaluate the same code for several different values of input. For example, suppose we'd like to write a program that applies the drinking age test to several different ages.

We could do this by simply copying and pasting our code in Spyder and running it several times. But this is inconvenient and bad practice: if we want to come back and change the repeated code, we'll have to do it in many places.

A better way is to write a *function* that, given any value of age, returns the appropriate result, and that we can use repeatedly.

```
def FUNCTION_NAME(PARAMETERS):
    # code block
```

Notice that like conditionals, functions require the use of a colon to start them and indentation to tell Python what is included within the function.

For example, for simple addition, we can do the following:

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

add(2.3,8)
```

This function takes two inputs `a, b` as *input* and gives their sum as *output*. This is done via the `return` keyword, which terminates the function and outputs the variables that follow it.

You can think of a function in Python as analogous to mathematics: a function takes (or not) parameters in input, does some operations, and returns (or not) a result. You can have a function to display a string:

```
In  [1]: def display_string(string_to_display):
             print(string_to_display)
```

Now, what if you wanted to allow the user to only provide one number, in which case one is added to it? You can do that by specifying a **default value**. The second input is then optional to the user.

```
In  [3]: def add(a, b=1):
             return a+b

In  [4]: add(2, 3)
Out [4]: 5

In  [5]: add(2) # In this case, b will be set to 1 by default.
Out [5]: 5
```

4.4.1 Exercise

Rewrite your code for the drinking age inside of a function can_drink that take as a parameter the age, prints 'can drink' if the age is higher than 18 and prints the remaining years to go before the person can drink otherwise.

4.4.2 Solution

We need to wrap the code written earlier inside a function:

```
def can_drink(age):
    if not isinstance(age, int): # This is new!
        age = int(age)

    # Test if age is higher than 18
```

```
    if age > 18:
        print('you can drink!')
    else:
        remaining_years = 18 - age
        print('Still {} years before you can.'.format(remaining_years))
```

4.5 Loops and lists

We now know how to use conditional statements to control program execution. Another powerful control mechanism is iteration, which allows us to take advantage of computers' tireless ability to repeat computations. We'll look at *while* and *for*-loops for iteration.

4.5.1 While loops

We'll first look at *while*-loops. Like a conditional statement, a while-loop begins with a test. If the test value is `True`, the program executes a block of code, but then returns to check the test again until the test evaluates to `False`.

```
while CONDITION:
    # code block
```

Here's a simple example of a loop:

```
x = 5
counter = x
while counter >= 0:
    print(counter)
    counter = counter - 1
```

Try the program out. What does it do? In short, the code binds the variable `counter` to the value of `x` (here 5). It then evaluates the condition for the loop: `counter > 1` which is `True`, so the program runs the code block inside the while loop, updating the value of `ans`. Importantly, it decreases the value of `counter`, which affects the loop condition. The loop is repeated until the condition returns `False`.

What would happen if we did not decrease the value of `counter`?

Note: if you get stuck in a long (or infinite!) loop in Spyder, click on the red square on the top right of the console window!

Consider another example:

```
x = 5
counter = x
ans = 1
while counter > 1:
    ans = ans * counter
    counter = counter - 1
print(ans)
```

What does this program do? It may be helpful to add `print`-statements of the different variables within the loop to understand what happens. What have we calculated?

4.5.2 Lists

We've so far mainly dealt with *scalar* data such as integers and floats. Often we're dealing with *sequences* of scalar data. A list is represented using square brackets in Python.

```
L = [0, 1, 2, 3, 4]
type(L)
```

We can access the elements of a list through their indices. The index of the first element is zero. Here are some ways to access a list: try them out!

```
L[0]
L[1:2]
L[-1]
L[:-1]
```

You can take a slice of a list l using the syntax l[begin:end:step]. Note that this goes from the begin-th element included to the end-th element excluded with a step of step.

We can further change the elements of the list and create new lists from existing ones.

```
L[0] = 4
L.append(99)
u = L + L
```

The length of a list is given by the function `len()`.

```
len(L)
```

Lists are flexible, and can contain different types of data, other lists, etc.

```
L = [1, 2, 4, 5, 'a', 5, [2,3]]
```

You'll be using lists constantly when writing programs in Python. See chapter 5.2 of the textbook for more details on how to work with lists.

We often use lists in looping. For instance, let's write code to find the index of the first instance of the number `5` in a list:

```
>>> l = [1, 2, 4, 5, 4, 5, 6]
>>> index = 0
>>> found = False
>>> while not found:
...     element = l[index]
...     if element == 5:
...         found = True
...     index += 1 # equivalent to index = index + 1
... print(index)
4
```

Here `found` is a Boolean variable that provides the test for continuing the loop. The loop procedes through the list until a `5` is found or the list is exhausted.

In a loop, you can use the keyword `break` to exit (break) the loop. For instance we can rewrite the previous example without updating the variable found:

```
>>> l = [1, 2, 4, 5, 4, 5, 6]
>>> index = 0
>>> while not found:
...     element = l[index]
...     if element == 5:
...         break # Leave the loop
...     index += 1
... print(index)
4
```

4.5.3 For loops

When writing algorithms, it is common for us to iterate over sequences like lists. Python provides a useful mechanism for this. A for loop works as follows:

```
for VARIABLE in SEQUENCE:
    # code block
```

A for loop goes through the variables in a sequence in order, executing the code block for each variable until the sequence is exhausted (or a `break` statement is used).

The sequence of variables is often generated using the built-in function `range`, which returns a sequence of integers based on its input. Specifically, `range(start, end, step)` produces all the elements from `start` to `(end - 1)` with a step of `step`. The default is `step = 1` and `start = 0`, so `range(4) == range(0,4) == range(0,4,1) == (0,1,2,3)`.

Here's an example of a simple for-loop:

```
x = 5
for i in range(x):
    print(i)
```

More generally you can use a for-loop to iterate through any sequence:

```
>>> l = ['2', 'a', 'hello']
>>> for element in l: # Loop through all the elements of l
...     print(element)
2
a
hello
```

Here we're going through the list `l` in order by simply using the keyword `in`.

5 Importing modules

Python comes with a large standard library, i.e. a collection of code implementing common functions. The module system also allows users to write their own code and distribute it easily.

To import a module you simply write `import module`. To use functions from the module the syntax is `module.function`. For example, there is a module called `math`. Go ahead and import it and try using the square root (`sqrt`) function.

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

Alternatively you can import directly a function from a module:

```
>>> from math import sqrt # We are importing only the function sqrt
>>> sqrt(2)
1.4142135623730951
```

6 Exercises: do as many as you can

6.1 Exercise 1: max, min and abs

Reprogram the following built-in functions (obviously *without* using said built-in functions):

1. write a function max_float that returns the maximum of two floats.
2. write a function max_list that returns the maximum value of a list.
3. write a function min_float that returns the minimum of two floats. Can you write it in one line?
4. write a function min_list that returns the minimum value of a list.
5. write a function abs_val that returns the absolute value of a float.

When you have done this compare your results to the built-in functions (max, min and abs) and check that you obtain the same results.

```
>>> max_float(2.3, 4.5)
4.5
>>> max_float(-2, -4)
-2
>>> max_list([1, 2.3, 4, -3.2, 3])
4
>>> min_float(2, 3)
2
>>> min_list([1, 2.3, 4, -3.2, 3])
-3.2
```

6.2 Exercise 2: golden ratio

Compute the golden ratio.

$$\varphi = \frac{1+\sqrt{5}}{2}$$

Hint: remember you can find the square root function in the math module.

6.3 Exercise 3: factorial

Right a function fact(n) that returns factorial of n = $n! = n \times (n-1) \times \cdots 2 \times 1$. (Remember that factorial of 0 is 1, $0! = 1! = 1$).

6.4 Exercise 4: pi

Compute an approximation of Pi using:

3.1. Newton/Euler's formula, using your factorial function:

$$\frac{\pi}{2} = \sum_{k=0}^{\infty} \frac{2^k k!^2}{(2k+1)!} = 1 + \frac{1}{3}\left(1 + \frac{2}{5}\left(1 + \frac{3}{7}\left(1 + \cdots\right)\right)\right)$$

3.2. the Wallis formula:

$$\pi = 2\prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Which one seems to converge faster to the correct solution?

Note that the sum starts at 0 while the product starts at 1!

6.5 Advanced topic: list comprehensions

In the section on loops you saw how to create a list in a for loop.

For instance, to create the integers from 0 to 9, you could do:

```
>>> l = []
... for i in range(10):
...     l.append(i)
>>> print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

However, another possible syntax, more concise and more *pythonic* (a slightly geeky way to say that the code follows the conventions and best practices of the Python coding community) would be using what is called *list comprehension*:

```
>>> l = [i for i in range(10)]
>>> print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can find much more on lists and list comprehensions in chapter 5.2 of the textbook.

NB: You could directly use the built-in function `range`. A subtlety about it: it returns a *generator*, ie a list-like object that you can only go through once. To obtain a list, you have to explicitly type-cast it into one.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.6 Exercise: list comprehension

Write two function that, given a limit N, $N \in \mathbb{N}$, builds a list of all the integers from 0 to N to the power of 2. For instance, for $N = 3$, your function should return [1, 4, 8].

The first version should use a for loop, while the second version should use list comprehension. As you increase N, which function seems to be the most efficient (fastest)?

Hint: You can use the `%timeit` command preceding a line of code to evaluate the time it takes to run.

7 What next?

You've now gone through most of the basic concepts and elements of Python that we'll be using in the course. These include *objects*, *variables*, different types of data such as *int*, *float*, *str*, and *list*, as well as statements to control program flow including *conditionals* and *looping*. You've also learned how to write a *function* to re-use your code.

We will come back to these concepts in the first lecture. Meanwhile, if you'd like to learn more about them, the textbook's chapters 2-4.1 cover much of the same material in a bit more depth. Note though that the book uses an older version of Python (2.7). Almost all code is interchangable between the two versions. However, there are some differenes: the one you will run into is that `print` works differently in Python 2.7:

```
print("hey") # Python 3 - the course and your Python installation
print "hey" # Python 2.7 - the book
```

Some further resources are:

- Beginners guide to Python (there are versions for non-programmers and programmers): https://wiki.python.org/moin/BeginnersGuide/NonProgrammers
- Another useful list of Python resources for all levels of experience and different application areas: http://www.fullstackpython.com/best-python-resources.html
- The official documentation: https://docs.python.org/3/
- The scipy lecture notes: http://www.scipy-lectures.org/