Programming Lab II
Handout 1

Antonio de la Vega de León, Bijun Zhang, Thomas Blaschke, Dr. Martin Vogt
`martin.vogt@bit.uni-bonn.de`

April 19, 2016

Starting remarks and Course organization

Programming Lab II focuses on learning Python as an alternative to Java, which was covererd last semester. The first part of the course loosely follows the second edition of the book *Think Python*. The book is open source and is available from `http://www.greenteapress.com/wp/think-python-2e/`. It can also be found in `\\bitsmb\groups\workshops\proglab2\` where all materials will be made available.

Course requirements

As a general rule you will receive handouts containing assignments on a weekly or biweekly basis that are to be completed within one or two weeks. Apart from the whitsun break there are two public holidays falling on a Thursday (Ascension and Corpus Christi); consequently, there will only be one class in those weeks. In these cases handouts will cover material for two weeks. Assignments have to be completed in the allotted time. They should not be handed in but should be shown to a tutor at the end of the class on Thursdays. If you were not able to finish on Thursday you still have the opportunity to present your assignments/programs on the following Tuesday.

Credit requirements

Individual assignments will not be marked. Instead, the assignments of a single handout will be judged as a whole and you will be more or less be awarded a pass or fail on that basis. You get full points if you were able to accomplish the majority of the assignments from the handout (no need to be p^{e}rf$_{c}$et). A pass will be worth 1, 2, or 3 points depending on how many classes were scheduled for completing a handout. There will be 22 classes (i.e. 22 points) this semester and you will need to achieve 16 points in order to pass the course. Again, it is not mandatory to do all exercises to get full marks for a handout. However, you should be able to demonstrate that you are in principle *able* to complete the exercises.

First remarks

What is Python?

a) A **python** is a constricting snake belonging to the Python (genus), or, more generally, any snake in the family Pythonidae.[1]

b) Monty **Python** were a British surreal comedy group that created Monty Python's Flying Circus, a British television comedy sketch show that first aired on the BBC on 5 October 1969.[1]

c) **Python** is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C (or Java for that matter).[1]

For the purpose of the course we will mainly focus on the last definition given above. However, the current logo of Python is inspired by the first definition and a lot of small examples of code you may find on the web or in books on the Python programming language are inspired by the second definition.

What is the Python programming language?

> Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, and classes.[2]

Let's take a look at the indidual characteristics:

Python is a programming language. Wikipedia defines a programming language as an artificial language designed to *communicate* instructions to a machine, particularly a computer.[1]

An artifical language is also known as a *formal language*. In contrast to natural languages programming languages are defined by a (relatively small) set of precise rules (the grammar) that define the formation of (syntactically) correct texts known as programs or source code. The program or source code is used to express a certain computation to be carried out by a machine and is defined by the *semantics* of the programming language. (Both of which are quite extensive fields in computer science.)

Programming languages can be very different in the way they express computations to be performed by a machine. However, a computer only 'understands' a single language natively known as the *machine code*, which on a most basic level can be represented a a sequence of 1's an 0's. (Different computer architectures understand different machine codes, the two most prominent in current consumer electronics being the Intel architecture for PCs/notebooks and the ARM architecture for mobile devices.)

Machine code is very hard for humans to write directly and high-level programming languages can be seen as a middle ground between natural languages and machine code. On the one hand, they are expressive enough to describe computations and complex processes on an abstract, conceptual level removed from the underlying hardware while, on the other hand, are rigorously defined having a well defined semantic (in contrast to natural languages, e.g. "time flies like an arrow, fruit flies like a banana") so that 'special' computer programs can be used to translate the source code of the high level language to the low level machine code language.

Python is interpreted, interactive. Python is a prime example of an interpreted language in contrast to Java, which is compiled. The difference is mainly how and when the source code is translated and executed on the machine. In a compiled language translation and execution

[1]Source: `en.wikipedia.org`
[2]Source: `https://docs.python.org/3/faq/`

are two clearly distinguished steps. In the translation step, source code is translated into object code resulting in a translated program that can be executed independently of the source code. E.g., the java compiler generates `.class` files from `.java` files. An interpreter does not have this explicit intermediate step. Instead, an interpreter takes the source code, translates it more or less line by line, and immediately executes it. For this reason, interpreted languages often provide an interactive mode allowing you to type in single statements, which are executed as soon as you enter return.

Python is object-oriented, has exceptions, and classes. These are characteristics Python has in common with Java. However, the philosophies of the languages are quite different so they are implemented very differently. Unlike Java, Python does not force you to write your program in an object-oriented matter, instead it is possible to be very productive in Python by avoiding class definitions simply using the classes the language and its extensions provides.

Python incorporates modules. Similar to the Java standard library Python comes with the Python standard library, an overview of which can be found under `http://docs.python.org/3/library/index.html`. These are organized in *modules* that collect functions and classes under a common *namespace*. In addition there are a lot of useful third party extensions like `biopython` for doing bioinformatics or `numpy` for handling numerical data.

Python has dynamic typing. Dynamic typing is one of the characteristics of Python that distinguishes it from the statically typed language Java. In Java you have to declare a variable and the type it has before you use it. In Python variables are neither declared nor is the type of a variable fixed. While in Java the type is associated with the variable (i.e. the variable name in the current scope) in Python the type is associated with the value.

Python does not have braces. Using curly braces (`{`,`}`) is a very common element in programming languages to organize the source code in blocks while on the other hand the typographical layout has no semantic meaning. That is, if you want you could write a whole Java program in a single line because the compiler mostly ignores whitespaces (i.e. space, tab, and newline characters) and only relies on curly braces and semicolons to structure the code. However, for (human!) readability the typographical layout of a program plays a major role and Java programs are typically typeset with one (simple) statement per line ending with a semicolon and blocks of code indented by a fixed amount of spaces determined by the nesting depth of the curly braces.

If the typographical layout of a program does not correspond to the syntactical hierarchy as defined by the program syntax this can be a cause for serious and hard to find bugs.

Python takes a different approach. Here, the typographical layout has syntactical meaning and defines how the code is organized. I.e., instead of using braces the layout defines how code is organized in Python

- newlines separate statements from one another, a semicolon is not needed.
- Indentation, i.e. the number of spaces/tabs a line is shifted to the right determines the code organization. E.g., consecutive lines indented by the same amount belong to the same block of code.

A lot of criticism of Python is based on its use of whitespace for organizing code and indeed it can be a source of confusion specially because spaces as well as tab characters can both be used to indent lines making code that mixes spaces and tab characters error prone. Some text editors, especially in Python IDEs, might be aware of this and will automatically translate tabs to spaces or vice versa. For editors like the *Crimson editor* or *Notepad++*, which might be installed on the Windows PCs, space and tab characters can be visualized and it is suggested that you do this if you use these editors.(**Warning:** Do **not** use *Notepad* or *Wordpad* for programming.)

Assignments

Preliminary Exercises

Ex. *Getting Python*
On the `\\bitsmb\groups\workshops\proglab2\` you find the zipped folder `pywin32` containing a full Python distributions for Windows. Copy and unzip it in your local home. The distribution folder contains two important links/scripts:

(a) `_python_session_` If you run this shortcut a new command line terminal is opened from which you can start the Python interpreter from the *command line*. Using a terminal like the command line is the traditional way of working interactively with Python. This is not a very comfortable way — especially on Windows. An interactive session usually consists of starting the Python interpreter by executing the `python` program. A special prompt appears (`>>>`) and you type stuff for the python interpreter to execute, hit return, and get a result on the screen on the next line, rinse, repeat... This is known as a Read-Eval-Print Loop (REPL) and is a common way for interacting with interpreted languages.

(b) `_python_notebook_` Because the standard REPL of Python is so useful but not very comfortable, the *IPython/Jupyter* project has made it its goal to provide a more comfortable way for interacting with Python. IPython provides several frontends for the user, notably a terminal-based, a GUI-based, and a browser based one. The browser-based frontend has developed into an independent web application known as Jupyter supporting many other programming languages. The supplied distribution contains a shortcut to start Python in this mode. It is suggested that you use Python notebooks for your interactive sessions. It will keep track of your session in notebook files that can be commented, edited, and extended.

Starting up the notebook will take a few seconds. It is set up to save notebooks in a folder called `_my_notebooks_`.

The (free) Python distribution is called *anaconda* and provided by a third party `https://www.continuum.io/`. The advantage of using this version over the locally installed versions of Python on each of the computers in the PC-pools is that you can easily install additional packages for the Python programming language like `biopython` or `nltk` using the `conda` package manager from the command line.

On your own PC/notebook you can also get a full version of the Anaconda Python environment from `http://continuum.io` already containing all of the available packages.

Ex. *The official home of Python*
The official home of the Python programming language can be found under `http://www.python.org/`. Here you can find 'official' downloads of Python for different platforms and the relevant documentation including tutorials.

Ex. *First steps* Work through the *First Steps* section below.

Ex. *Reading*
(*Note:* It is expected that reading of the material is done outside class hours and the time during class is used for doing practical assignments and asking questions to the tutors.)

Study chapters 1 to 3 of the book *Think Python*. You might want to use the Python notebook while reading through the book to try some things out.

Check your understanding by answering the following questions:

a) Name a conceptual difference between Python and Java.

b) Which kinds of basic instructions will you find in every language?

c) What three types of errors can occur in (Python) programs and what is the difference between them?

d) How does a 'Hello, World!'-program look in Java compared to Python?

e) What is the difference between a statement and an expression?

f) You should have seen a few Python error messages by now. What kind of error is a `NameError` in Python? What kind of error would it be in Java?

g) What are *keywords* used for in a programming language?

h) What types of objects have you encountered in the first three chapters? Hint: Practically everything, which has a name (except keywords), also has a type.

Programming exercises

You can solve the following exercises using Python notebook. for instance, you can create a single notebook for this week's exercise. You can organise your notebook by inserting *headings* and *markdown* text. See the documentation `http://ipython.org/ipython-doc/stable/notebook/notebook.html` or some newer tutorial like `http://bebi103.caltech.edu/2015/tutorials/t0b_intro_to_jupyter_notebooks.html` for details.

Ex.1. *Python as a calculator.*

> If you run a 10 kilometer race in 43 minutes 30 seconds, what is your average time per mile? What is your average speed in miles per hour? (Hint: there are 1.61 kilometers in a mile).

Ex.2. *Some typical errors*
Enter the following code in the notebook:

```
width = 17
height = 12.0
print("Area:",width * height)
```

Try the following lines:

1. `print("Area: ",width * heigt)`
2. `print "Area: ",width * height`
3. `print("Area: ",width * width)`

Examine the outputs and decide in each case whether you got a syntax, runtime, or a semantic error. Now try as a single input:

```
width␣=␣17
height␣=␣12.0
␣␣print("Area:",width␣*␣height)
```

by inserting some spaces (␣) before the print function at the beginning of the line. What kind of error do you get?

Ex.3. *Variables and types*
Define the following variables

```
width = 17
height = 12.0
delimiter = '.'
```

and print the the *value* and the *type* of the following expressions:

1. `width//2`
2. `width/2`
3. `height//3`
4. `delimiter * 5`

Did you get the results you expected?

Ex.4. *Functions* Python provides a built-in function called `len` that returns the length of a string, so the value of `len('allen')` is 5.

Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.(ex. 3.3 of *Think Python*):

```
>>> right_justify('allen')
                                                                 allen
```

Test `right_justify` on the following lines

```
We are the Knights who say, Ni!
Bring us a shrubbery!
You must bring us...another shrubbery!
And now you must cut down the mightiest tree in the forest with ... a herring!
```

Are all lines properly aligned?

- Save your function definition in a file called `just.py`.
- Remember that you can run the script in your notebook using `%run just`.
- Running this script in a new notebook will define this function so that it can be used later on.

Ex.5. *Turtle World*
Work through chapter 4 of the book *Think Python* following the instructions as you go along.

Caution: The Python turtle module does not work too well with the Python notebook. You can still use it interactively but repeatedly opening turtle windows might not work. In this case you need to *restart the kernel*.

(a) Write a function `hexagon(t,length)` that draws a regular hexagon where each side has length `length`.

(b) Write a function `benzene(t,radius)` that draws a benzene ring by combining the function `hexagon` from the previous exercise and `circle` from the book.

(c) Compile all the function definitions from the book and this exercise in a single script named `figures.py`. In a new notebook you can now use

```
In [ ]: import turtle
        from figures import *
        bob = turtle.Turtle()
        benzene(bob,100)
```

Ex.6. (*Optional*) Write a function with suitable parameters that is able to draw flowers as shown in Figure 4.1 of *Think Python*.

First steps in Python

The course will focus on Python 3. Although Python 3 has been introduced more than seven years ago, for a long time Python 2 has remained the more popular of the two versions. A major factor in this was that for a long time popular 3rd party products (like BioPython) required Python 2. By now, most important packages are also available for Python 3, which is the version we are going to use for the course. In Python 3 the language changed in some details, a consequence of which is that Python 2 programs will not run under Python 3. The changes in syntax and semantics are not very big, so there is not much difference in learning the language. One of the most obvious changes when starting out in Python are that the `print`-*statement* of Python 2 has been changed to a `print`-*function* in Python 3 and that the way that integer division works has been changed. (see below).

Start the Python notebook by double-clicking `_python_notebook_` in your Python distribution. A new tab will open in your browser with the location `localhost:8888/tree` that represents the toplevel interface. Here you can either select a Python notebook file (`.pynb`) or create a new notebook file. These will be saved in a folder called `_my_notebooks_` inside the `pywin32` folder. For now, the important tabs are *Files* for opening and creating notebooks and *Running* for navigating and closing current notebooks. Create a new notebook by selecting *Python 3* under the heading *Notebooks* in the selection box *New* on the top right of the web page. This will open a new tab with a an empty notebook session.

On the top, the name of the notebook is displayed as `Untitled`. You can change it to `First Steps` if you want to by clicking on it. Notice that the top of the page has a menu structure and a toolbar with commonly used operations. The main part of the window consists of input boxes and output. Normally, input will consist of some Python code and the output will be the result of the execution of the Python code. Input lines are prefixed with `In [#]` and the corresponding output with `Out [#]` where `#` represents an increasing number.

Click on the text field beside `In [ ]` to enter or edit code there. Try typing `print("Hello world!")` and press the Enter key in combination with the Shift key. (You use <Enter> in the input field to input multiple lines of code and you use <Shift>-<Enter> to execute the code.)

Numbers and Expressions

In the simplest case you can use the Python interpreter as a calculator. Try this:

```
In [ ]: 2 + 2
Out[ ]: 4

In [ ]: 4711 + 22
Out[ ]: 4733

In [ ]: 2 + 4 * 3
Out[ ]: 14

In [ ]: 1.5 + 0.7
Out[ ]: 2.2
```

There is one caveat here. Python distinguishes between different kinds of numbers. Namely integer numbers, which are not allowed to have fractional part, and floating point numbers, which can be fractions. Numbers without a decimal point are integers and numbers containing a decimal point are floating point numbers. So Python sees `42` as the integer 42 and `42.0` as the floating point number 42.

This is not an arbitrary decision by Python but is also reflected by the computer hardware. The central processing unit (CPU), which is responsible for all calculations, has a dedicated unit especially for doing floating point calculations and one for doing calculations with integer numbers. With integer numbers calculations can be done very fast and are always exact. Floating point numbers are not always exact and are susceptible to rounding. Due to the *binary* format used to store numbers even simple *decimal* numbers might not be exactly representable. Try:

```
In [ ]: 0.1 + 0.2
```

Now try:

```
In [ ]: 6 / (1 - 3/4)
```

Is this the result you expect? In languages like Java and the older Python 2 version, the result would have been 6 instead of 24 because the division of two integer numbers would always result in an integer number making 3/4 == 0 instead of 3/4 == 0.25.

> Side note.
> At one point the makers of Python decided the Java way is really not the way the language should work and they decided to change that behavior. But changing a behavior like that means that programs relying on Java-like behavior for division would give different results in the new Python version. This is one of the fundamental changes that make the Python 3 language different from Python 2.

To get integer division (i.e. like in Java) `//`. Another useful operator is the remainder operator `%`. While `//` gives the integer part of an division `%` gives the remainder of a division:

```
In [ ]: 11 // 3
Out[ ]: 3

In [ ]: 11 % 3
Out[ ]: 2

In [ ]: 2.3 // 1.1
Out[ ]: 2.0

In [ ]: 2.3 % 1.1
Out[ ]: 0.1
```

The final mathematical operation is the exponentiation operator `**`.

```
In [ ]: 2 ** 3
Out[ ]: 8
In [ ]: (3 + 4) ** 3
Out[ ]: 343
```

Python can also handle very large numbers

```
In [ ]: 12345678**9
Out[ ]: 6662458388479360230805308787387369820914640828074410829911019008
```

These numbers are larger than what a CPU can natively handle.

You can save your notebook by clicking on the disk icon of the toolbar. It will be saved under the name you have given it in the folder from where you created it. If you close the browser tab and go back to the top level page `localhost:8888/tree` you can see a `Running` tab next to the `Files` tab. If you click on it you should see your notebook listed and you can open it again by clicking on it or closing it using the shutdown button. Once you close it using the shutdown button, open it again from the `Files` tab.

A First Python Script

We will call a (Python) program that is stored in a (single) file a *script*.

> Using a Text Editor.
> Unlike Word or other office programs, text editors are used to edit *plain text files*. These consist of just the characters that make up the text and have no additional information like which font is used, how the text is formatted (except for spaces and line breaks, which are just (special) characters). On

Windows you can run the notorious `notepad.exe`, which is a very basic text editor or use a more fancy one like the 'Crimson Editor' or 'Notepad++' . Although program source files are just plain text files advanced text editors are able to highlight the syntax of different programming languages usually using different colors and help in the correct formatting of your programs. A lot of free text editors are available that 'understand' the syntax of many programming languages. You might want to try a few until you find one you like. Alternatively, you can use the browser-based editor integrated in the Jupyter front-end.

So far we have been using Python interactively typing line by line and getting the result of each line separately. The interactive mode of Python is very handy because it lets you try things out and immediately see if they work the way you intended.

But now we want to write our first real python script in a separate file and let the Python interpreter execute the script. For our first program open a *text editor* and create a file containing the text:

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

Save the file under the filename `hello.py` in the directory where your notebook is saved.

Now open your notebook again and type

```
In [ ]: %run hello
```

This will run the script and show the output.

Windows file extensions.
You will probably be working with files having quite different extensions, some of which may be associated with programs on the computer. By default, Windows does not show extensions of file types that are associated with some program. In order to make Windows do that:

1. Open the Windows Explorer (Windows button + E)
2. From the menu, select `Tools->Folder Options`
3. Go to the View tab
4. Uncheck the option "Hide extensions for known file types"

Now let us make the script a little more interesting. Use the text editor to create a new file:

```
name = input("Type in your name: ")
print("Hello, " + name)
```

Save the file under the name `hello2.py` and type

```
In [ ]: %run hello2
```

Notice that the Python program writes `Type in your name:` and then appears to wait for something.

The *function* `input` expects some input from the keyboard. If you type your name and press return. The *return value* of the function `input` is the *string* of letters you typed in. This value is bound to the *variable* `name`. In the next line the `print` function writes a string to the console that consists of `Hello,` and whatever you typed in.

For a more comprehensive documentation of the Python notebook features see: `http://ipython.org/ipython-doc/stable/notebook/notebook.html`.

Tips for using Crimson Editor or Notepad++

Crimson editor and Notepad++ are two editors installed under Windows that are useful for writing your Python scripts. They are able to 'understand' syntactical elements of Python so that keywords and other constructs of the languaghe can be highlighted by different colors.

A speciality of the Python language is that so-called whitespace characters like `<space>` or `<tab>` are important for structuring programs. This is in contrast to a lot of other programming languages that frequently use braces like `{` and `}` to this end. Because of this it is very useful to visualize space and tab characters while writing and editing your scripts. You can achieve this

- in Crimson Editor by selecting in the menu View→Others→Show Spaces and View→Others→Show Tab Characters,
- in Notepad++ by selecting View→Show Symbol→Show Whitespace and TAB.