Starting

Read: *Chapter 1, Appendix B from textbook.*

- There are two ways to run your Python program using the interpreter[1]: from the *command line* or by using *IDLE* (which also comes with a text editor; more about this later). Here we will use the command line version. You can start by typing

  ```
  >python
  ```

  You should get something like the following:

  ```
  Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
  [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
  Type "help", "copyright", "credits" or "license" for more information.
  >>> ■
  ```

  Notice the Python version (we are using Python 3), and the Python prompt >>>
- Our programs are a collection of statements[2]. Type the following commands:

  ```
  >>> print(“Hello!”)
  >>> print(2+3)
  >>> print(“2+3=”, 2+3)
  ```

- You can always get some help from Python about a topic by typing

  ```
  >>> help(topic)
  ```

  Type

  ```
  >>> help(print)
  ```

  Press the Q key to continue with the interpreter. Most probably you will not understand much from what help() outputs (!), but our aim is to be able to understand most of it by the end of this course. For now, the Python tutorial at

  http://docs.python.org/3.3/tutorial/

  is of much more help at this point.

---

1 Throughout this course, when you see a term in red font, you should look it up in your book, either by studying the corresponding chapter, or by using the **Index** starting at page 503. Hopefully you will do **both**: you encounter the term for the first time in the corresponding chapter, and afterwards, every time you need to refresh/clarify it again, you look it up in the index. In your lab documents you are going to get a quick definition of terms; but they are *quick* and *dirty* definitions, i.e., they are *too short* and *too incomplete*. Here is the first one: an interpreter is an interactive environment for executing commands.

2 A complete executable command.

- print() and help() are examples of functions[3]. Type the following:

```
>>> def hello():
        print("Hello!")
        print("Computers are fun!")
```

This piece of code is called a **function definition**. If you failed (Python interrupted your typing with a message), then here are some common mistakes that all of us make all the time:

- You forgot the '**:**' at the end of the first line.
- You didn't follow the **indentation** you see in the example. Does your code look like this?

```
>>> def hello():
        print("Hello!")
      print("Computers are fun!")
```

Or like this?

```
>>> def hello():
    print("Hello!")
    print("Computers are fun!")
```

- You have some other syntax[4] error, such as quotes (") missing.
- You did everything right, but you haven't pressed ENTER twice to indicate that you are done typing your program (an empty line tells Python that you are done typing). Well...what are you waiting for? Go ahead and press it again!

Indentation[5] is extremely important in Python. It is the only way for the interpreter to recognize that a group of statements go together. Other languages put such groups within parentheses, brackets, etc. (for example, C uses brackets {...}). So, how many groups of statements do we have here? (that's right, we have two: line 1 (one group) and lines 2-3 (the second group)). The TAB key is very useful in Python. Since most of us learn better from our mistakes, type the following *as you see it*. What do you get?

```
>>> def hello():
        print("Hello!")
      print("Computers are fun!")
```

- You have just defined the new command hello(). We call such commands *functions*. Type

```
>>> hello()
```

3 A new command (defined by us) that executes a sequence of statements (also defined by us).
4 The rules for writing correct programs.
5 From the verb *indent*: To set back (from the margin of the column of writing or type) the beginning of (one or more lines), as a means of marking a new paragraph, of exhibiting verse, etc.; to begin (a line or a succession of lines) with a blank space. (Oxford English Dictionary)

This is called a **function call**. Note that your new function/command is executed only when you call it, not when you define it. And **you can only call what you have already defined**. What do you get? (If you still get errors or not what you would expect, rewrite the program.)

- The general pattern of a function definition is the following:

```
def function_name(<arguments>):
        <statement 1>
        <statement 2>
        ...
        <statement n>
```

The first line defines the function name and its list of arguments <arguments>. Our hello() function has no arguments. The group of <statement>s is the code that will execute every time we call (correctly) the function. Type the following:

```
>>> greet(“John”)
```

You should get something like the following:

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'greet' is not defined
```

The important line is the last one; it contains the interpreter's explanation of what your error is. In this case you have tried to use a **name** (greet) that has not been defined. **You can only use (call) what you have already defined.** Let's define the greet function by typing:

```
>>> def greet(person):
            print(“Hello”, person)
            print(“How are you?”)
```

The function greet has one argument named person. For now, think of this argument as a placeholder every time it occurs in your <statement>s. This placeholder must be filled in with 'something'; this 'something' is a value for the argument you provide when you call the function. By typing the following call to greet you provide the value “John” to argument person:

```
>>> greet(“John”)
```

What do you get now? Where did the quotes (“) in “John” go? Try the following:

```
>>> greet(John)
```

What do you get now? It should look something like the following:

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'John' is not defined
```

Note that the two calls treat `John` differently: When you write `"John"` you define a string[6]; its value is what is between quotes (“). When you write John you don't define a string, but you are using a name; and, as we know, y**ou can only use what you have already defined.** Try

```
>>> greet('John')
```

No error! In Python, you can define strings either within single (') or double (“) quotes. Try

```
>>> greet
```

You should get something like this:

```
<function greet at 0x100709cb0>
```

This line is not indicating an error; it informs you that `greet` is a function whose definition exists, i.e., it is in memory starting at memory address `100709cb0` (in hexadecimal). Don't worry more about this right now; it is enough to know that `greet` has been defined and the interpreter knows where to find its definition. Just to verify that hello is (still) defined for the interpreter, try

```
>>> hello
```

You should get a similar output, but with a different memory address (obviously, since the definitions of two different things cannot be in the same place).

- Now that you have become a function guru, what is going on when you typed the following?

```
>>> print(“Hello!”)
>>> print(2+3)
>>> print(“2+3=”, 2+3)
```

That's right! `print` is a function whose argument values are `“Hello!”` in the first call, `5` in the second call, and `“2+3=”, 5` in the third call. A few strange things are happening here:

  - We have never defined `print`. But y**ou can only use what you have already defined. L**et's find out where the definition is:

6 A sequence of characters.

```
>>> print
```

You should get something like the following:

```
<built-in function print>
```

A **built-in** function is one whose definition comes with the interpreter (think of it as the interpreter defining its own functions and names before you even get your first prompt).

- `print` seems to take a varying number of argument values (here either one or two, but you can give it as many as you like). Does it mean that it has a varying set of arguments in its definition? Not exactly. We will explain this later, but for now try

```
>>> help(print)
```

This tells you how print is called:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The three dots indicate an arbitrary number of values, before the last four (which themselves are optional).

- Note that instead of 2+3, `print` is passed the value 5, i.e., before passing the value to the function argument, Python evaluates[7] the expression[8] 2+3 to its final value 5.

- To finish your session with the command-line Python interpreter type

```
>>> quit()
```

- Start the command-line interpreter again and look for function `greet` – notice that its definition has disappeared from the memory. Exit the interpreter again (using `quit()`).
- Start IDLE. You should get one window with the interpreter running; **ignore it**. If you open a second window, that is a text editor. We will use the text editor to write our programs. Each program we write will be mainly a collection of function definitions. We will save each program in a file (called **module** in Python). Then every time we need to use our functions, we can load(**import**) them from that file; no need to rewrite everything.
- Type the following in the text editor:

```
# File: first.py

def greet(person):
        print("Hello", person)
        print()
        print("How are you?")
```

7 Calculates the final value.

8 A combination of operands (=anything that boils down to a value, e.g., numbers like 2 or 3) and operators (=an operation applied to one, two, three, or more operands, that combines them to compute a new value, e.g., +, -)

- Save what you have written as a file named first.py in the directory where you run the command-line interpreter.
- Start the command-line interpreter (in the directory where first.py is).
- Try to find (and fail) function greet:

```
>>> greet
```

- Import module file first.py by typing:

```
>>> import first
```

Now try the following:

```
>>> first.greet("John")
```

What do you get? The explanation is quite simple: When you import a module, it is like you write the lines of the module one-by-one in the interpreter. The only difference with your previous greet is the extra print() statement (which prints an empty line), and that you have to specify the module that contains greet (by typing first.greet instead of just greet). Can you see why you need this extra 'dot' stuff? (*Hint: what if I want to import many modules, some of them with their own function greet in them?)*

- Probably you've noticed the first line of first.py, the one that starts with #. The interpreter ignores all lines that start with #, so you can use it to insert **comments** in our programs. Comments are ignored by the computer, but are extremely useful for other humans to understand what a program does.
- Go back to the text editor window and start a new text file.
- Type the following in the text editor:

```
# File: chaos.py
# simple program that illustrates chaotic behaviour

def main():
        print("Chaotic function")
        x = eval(input("Enter a number between 0 and 1: "))
        for i in range(10):
                x = 3.9 * x * (1-x)
                print(x)

main()
```

- Save this module in a file chaos.py (in the directory where you run your command-line Python interpreter).
- Go to the command-line interpreter and import this module. What do you see? Instead of

getting the Python prompt after `import` like before, it looks like function `main()` is run. How? Recall that importing a module is like typing it line-by-line in the interpreter yourself. Look at the last line of the module: it is a **call** to `main()` (which, of course, **follows** the definition of `main()`). We go over the rest of the module line-by-line to just mention terms we will explore in depth in later times; the book gives more details in Section 1.7.

- Lines 1-2 are comments.
- Line 3 starts the definition of main. Notice that `main` here is defined without arguments. Usually all Python programs have a function `main` in them, and this is the function that is called when we want to run the program.
- Line 5 is rather complex. It is an **assignment** statement: it assigns the value calculated in the Right-Hand Side (RHS) of the '=' to the **variable** `x` in the Left-Hand Side (LHS). The value in the RHS is the result of function **`eval`** which takes one argument value; here this value is the result of function **`input`** (which actually is what you type when asked "Enter a number between 0 and 1: ").
- Line 6 starts a **loop** that executes the statements in lines 7-8 (note the indentation; this is the **body** of the loop) 10 times (as dictated by `range(10)` in line 6). This is equivalent to replacing lines 6-8 with 10 copies of lines 7-8 one right after the other. Each copy of line 7 changes the value of variable `x` (in LHS) to the value computed in the RHS (using the current value of `x`). The `print(x)` that immediately follows prints the new value of `x`.

- Get some extra information about `eval` and `input` from the interpreter (even if you don't understand it right now).
- Call function `main()` again **without importing the module again**. *(Hint: Did you forget the 'dot' part?)*

Practice problems

1. Try to do all the book problems.
2. Use IDLE to open a text editor window. In it, define a small function `myname()` that prints your name. Save this file as `name.py`. Find out whether this file is in the directory from which you launch the Python command-line interpreter. If it is not, then go back to the text editor window, and try to find the "Save as..." option in the window menu. Click on that, and this should open a dialog window where you can specify the name of your file, and the directory where it will be saved; you should pick the directory where you call Python from. Check again whether the file has been saved in this directory. If not, **ask for help! It is extremely important for the rest of the semester to be able to save and load your files from and to any location you want!**
3. Start the command-line interpreter and import your `name` module and then call your `myname()` function. Did you forget the "dot" part?
4. Repeat (1) & (2) above, but now your function must take a parameter, and print the value of this parameter. *(Hint: Look at what the `greet` function above does.)*
5. Repeat (1) & (2) above, but now your function must take a parameter, and print the square root of the value of the parameter. Here, you will need to import the `math` module typing

```
import math
```

in order to use function `math.sqrt(x)` to calculate the square root of a number x. But where do you type the `import math` command? There are at least the following options (try them all, to see what happens, and maybe come up with some of your own possibilities):

1. In the command line, *after* you import your own module and call your function.
2. In the command line, *after* you import your own module but *before* call your function.
3. In the command line, *before* you import your own module and call your function.
4. In your own module, at the beginning (*before* you define your function).
5. In your own module, *after* you define your function.

Which one of these options seem to be the best for you and why?

6. Type, save, and run `chaos.py` from the text (if you haven't done so yet). Change your program from (4), so that it also has a `main()` function, in which you ask the user for a positive number x, and then it calls your function from (4) with this number x as the parameter value.
7. Write a loop that prints all integers from 0 to 100.
8. Write a program that asks the user for a number n, and prints n consecutive integers, starting from 0. *(Hint: Make sure that you print n numbers, not n-1 or n+1.)*
9. Write a print statement that prints the numbers (3.67/5.43) and (8.88+23.23) separated by the string "...is not..."