

Python for Artificial Intelligence

Stefano Benedettini

Andrea Roli

Artificial Intelligence LM 2010

Contents

1	Python hints	1
2	Python Boot Camp	2
2.1	Native Data Types	2
2.2	Functions	4

Topics

- Build-up some experience on `Python` language

1 Python hints

- Always keep your `Python` interpreter ready
- Always keep your documentation ready as well
- For now, *no* fancy IDEs with autocompletion, code assistance, automatic testing, project management or the latest bells and whistles: any text editor with syntax highlight and smart indentation will suffice
- If you are not sure about the behaviour of a piece of code, try it interactively in the interpreter
- Always try to guess the expected behaviour of a code snippet: it helps you reason about the language
- A very effective way of learning is the “What if?” approach: whenever you examine a piece of code, think about possible variations and try to predict the outcomes
- Don’t forget about useful `help` and `dir` functions in the interpreter

- For now, keep all your scripts in a single directory: you will learn about packages, search path and module details later (on your own actually, because it's boring stuff)

To get you started:

```
chmod u+x script_file.py
```

To make a script executable. Remember to put a shebang (`#!/`) to indicate the program to be used to run the script (usually `/usr/bin/python`).

```
python script_file.py <command-line-args>
```

Another way to run a script file.

```
if __name__ == "__main__":
    do_stuff()
```

Statements inside this block will be executed when enclosing module is run as a script.

```
help(str)
help(str.join)
dir("a string") == dir(str) # is True
```

Remember to use `help` function to display docstrings of classes and functions. Moreover, `dir` returns the list of all attributes of an object.

2 Python Boot Camp

This collection of exercises refers to material contained in the Official `Python` Tutorial available in standard documentation and Dive Into Python.

Before diving into the exercises, make sure you understood the language basics. Exercises will cover every aspect of the language, therefore do not expect very focused exercises on a specific language feature. We will follow a generalist approach: you should know a bit of everything before starting and you are encouraged to dig into details as the need should arise.

Exercises will roughly cover chapter 3 to 8 of the Official `Python` Tutorial and chapters 2–4 and 6 of Dive Into Python. As a guideline, skim through chapter 6 of Official `Python` Tutorial and make sure you read chapter 7 as soon as you got a grasp of functions and control structures, because file handling is introduced.

2.1 Native Data Types

In this block of exercises you will familiarize with powerful `Python` built-in data types. Besides number and strings, which you will certainly figure out by yourselves, `Python` provides *lists* (type `list`), *dictionaries* (type `dict`), *tuples* (type `tuple`) and *sets* (type `set`). Syntactic sugar is available for list, dictionary and tuple literals (also sets in 3.x).

1. Write a script that keeps asking user for numbers and collects them in a list. The script must cycle through these instructions until user inputs *"quit"* string. The script must then output the average of supplied numbers.

Highlights:

- Browse list documentation
- Examine `len`, `raw_input` and type conversion functions
- You can manage a bad input string two ways: exceptions or explicit check. Catch exceptions like this (or look it up in a tutorial, nothing different from Java anyway):

```
try:
    code_to_watch()
except NameOfException as exceptionVar: # likely ValueError in your case
    rescue_code()
```

2. Write a script that asks user for two numerical vectors of the same length. User must be able to provide a vector on a single line, i.e., she should not be forced to input each vector element by element. Vector representation is up to you. The script must return the euclidean distance of the vectors. Provide an implementation using:

- (a) a simple `for` loop;
- (b) functional programming tools like `map` or `reduce`;
- (c) list comprehensions.

Highlights:

- string methods
- `zip`, `map`, `reduce` functions

3. Write a script that keeps asks user for a number until she inputs a *"quit"* string. The script will determine whether the input number is prime. If it is, it must save it in a list. If user provides the same prime number twice, the script must notify it. In the end, the script must display all input prime numbers in ascending order.

Highlights:

- More list methods
- A small function (nothing you can't handle)
- `sorted` function

4. Write a script that reads a matrix (a list of lists) from a file. You decide data format.

Highlights:

- File manipulation

- Lists of lists
5. Write a word counter in **Python**. The program should expect a file name and return, line by line, the occurrences of each word in the given document. Suppose that a word is a sequence of characters separated by a space character, like space, tabulation or newline. Ignore capitalization. **Highlights:**
 - File manipulation
 - Optionally, you can look into **with** statement
 - Dictionaries
 6. As a variation on the last exercise, write a letter counter. The program should do the same thing as the previous exercise, only to letters instead of words. Consider 26 letters English alphabet and ignore capitalization and space characters. In the end, the program should return each letter frequency and compute Shannon entropy. For those who don't remember:

$$\mathcal{H} = - \sum_{c \in \Sigma} p(c) \log p(c)$$

Where Σ is an alphabet and symbol probability p can be approximated by frequency. Can you do the same for digrams (groups of 2 consecutive letters) and trigrams (groups of 3 consecutive letters)? Is entropy higher or lower? **[advanced]** **[optional]**

2.2 Functions

1. Write a script that reads two matrices from two files and computes:
 - (a) Matrix sum
 - (b) Matrix difference
 - (c) Element-by-element product
 - (d) Matrix product

Provide implementations that employ **for** cycles and list comprehensions. Knowing **Python** functional programming features, try to generalize matrix sum, difference and element-by-element product.

2. Write a function that computes the n^{th} element in Fibonacci sequence. Do you think a recursive implementation is feasible? Try it out.
3. Write a generic function that takes a number n and a function implementing a mathematical sequence x_i and prints the *series* at index n , that is: $\sum_i^n x_i$. Apply to the following sequences:

(a) $x_n = q^n$ for some $0 < q < 1$

(b) $x_n = (-1)^{n+1} \frac{1}{n}$

(c) $x_n = \frac{1}{n^2} \sin 2n + \frac{1}{i^2} \cos 2n$

(d) $x_n = \frac{1}{n^s}$ where $s \in \mathbb{C}$

4. **Python** is often used for batch text processing. Examine the dataset file "*cvrp-simple-100-50.txt*". Those you see are the results of a metaheuristic algorithm on a routing problem. Data is organized in tabular form, one experiment for each row. The first two lines are garbage. Row format is the following:

- instance name
- solution value (is a minimization problem)
- runtime to returned solution (in seconds)
- total runtime (in seconds)
- iterations to solutions
- total iterations
- number of vehicles used in best solution
- number of vehicles required by problem instance

If number of required vehicles doesn't match with number of used vehicles, solution is considered *infeasible*. For each problem instance, the algorithm was run 5 times.

- (a) For each instance compute the best solution returned in the 5 runs and the average solution quality
- (b) Compute average and total time for each problem instance
- (c) Sum up the previous statistics across all instances, that is: sum of average solutions, sum of best solutions, sum of average times, total time of the experiment
- (d) Compute the ratio of feasible solutions

Present all the results in readable tabular format.