1 Introduction

This documents contains a semi-detailed syllabus and day by day content for a python course developed at Sinclair.Bio oriented towards biologists and Big Data in genomics. If your organization is interested in setting up such a course with us, things can be customized and modified depending on your needs.

2 Why learn programming?

The last decades have brought many changes to all the domains of science and technology. One trend that is steadily increasing in importance is the necessity of using large quantities of data to answer scientific questions. For instance, the field of physics has almost entirely switched to data-driven research and, indeed, students in physics learn programming very early nowadays. An other field that has been heavily affected is that of biology. There, "Big Data" is also changing the fundamental ways in which science is conducted, the cost of DNA sequencing having dropped by five orders of magnitude in the last ten years. Yet, not all scientists are ready or have been trained for this sweeping change.

Everywhere, enormous amounts of data are now being produced and the computer operators required to analyze and process them have largely become the bottleneck in many research programs. Sadly, the computational skills needed are often not included in the standard university curricula, yet many scientists are now exposed to such big data and spend half or more of their time behind a keyboard instead of in the laboratory. In this course, we will give the practical knowledge that a scientist needs to address the common challenges of large-scale data analysis. This course won't make them professional information technology specialists, but it should make the participants literate enough to solve a large portion of their problems by teaching them how to program in a modern, widely-used, efficient and user-friendly language: python. They will also a learn a bit about programming tools around python that can help them organize and reuse.

In essence, we see many of our peers that are either stuck at a low level of proficiency and only venture into Excel for basic problem solving or, in the second case, are a bit more proficient but have been hired as de-facto IT scientists while lacking hard programming skills. After this course, the participants will become versed in python and they will be able to solve day-to-day computational problems by themselves. At the same time, the heavier users will be able to produce more readable, maintainable and shareable code, while automating large parts of their analysis and becoming leaps and bounds more productive.

Programming courses aimed at the modern scientist that missed this type of training in his undergraduate path are scarce. Some courses do exist but they tend to be much shorter and devote, in our opinion, too little time to hands-on exercise which are critical to the learning of a new skill.

3 Why learn Python?

There is a plethora of programming languages that have been developed over the last fifty years and the variety of choice can seem overwhelming at first. We should maybe start by asking ourselves: why should we even focus on only one language and not learn all the best ones? Well, the answer to that is quite simple. Firstly, it takes a significant amount of time investment to learn the modalities and syntax rules of a programming language, even for the ones that are designed to be quickly assimilated. Secondly, your productivity will be much greater if you have a deep mastery of a single multi-purpose programming language when compared to the situation where you have but a surface knowledge in many different programming languages.

It is therefore optimal in the context of answering scientific questions to select one particular language and to become good at it. The language should be modern, widely used (i.e. large community of support), easy to learn, and must be able solve a wide-range of problems (i.e. not a domain specific language.).

In addition, when choosing a language, there is a balance to be made between the speed of execution of the final product in terms of processor-hours and the time it takes to create the final product in terms of programmer-hours. Languages that are "close to the machine" such as Assembly will create programs that can run extremely fast and in an extremely resource efficient manner, however they will take months or years for the programmer to create. Languages that are "far from the machine" and are abstracted away from the bare metal will run in a slower fashion but will enable the programmer to create them in minutes or hours. The current situation in scientific computation is that most of the challenges commonly faced can be resolved without the need of highly efficient programming. It is thus desirable to choose a high-level or "scripting" language to promote the best productivity. In other words, letting your script run a couple hours or days is never a problem and you can work on other things in the mean time.

4 Learning Outcomes

The main goal of the course is to provide sufficient knowledge in the craft of programming to enable scientists to solve the easy and intermediate computational problems they will be confronted with independently. The course will give them a intermediate practical proficiency in the python programming language. After the course the participant should be able to:

- Write simple and intermediate programs in python to process, filter, clean, analyze, and visualize scientific data.
- Ability to automate much of the computational tasks that are used in their day-to-day work.
- Understand the universally used paradigm of object-oriented programming as it is implemented in python.
- Understand the necessity and advantage of using test-driven development.
- Understand the best practices in python programming, such as the advantages of style guides.
- Basic proficiency with the revision control solution that is "`git`" to archive and distribute the programs or scripts created.
- Acquire experience with understanding and quickly debugging errors within the code.
- Ability to read and understand programs written by one's pears, to review them or modify them.

5 Duration and time plan

The course will run over a month and be split into two parts, one basic part and one advanced part. The second part will be the natural continuation of the first part and will be focused around the personal project. The participants can inscribe to only the first part, only the second part, or to both.

The basic part would run over the course of two weeks and contain six half days as shown in figure 1. A half day would include four hours and a half running from 13h00 to 17h30 with

three coffee breaks included. Courses would be split evenly between lectures/seminars and exercise/practical sessions.

	Monday	Tuesday	Wednesday	Thursday	Friday
Week 1	Afternoon	Afternoon	Afternoon	no course	no course
Week 2	Afternoon	Afternoon	Afternoon	no course	no course

Figure 1: Schedule for basic part

The advanced part would include three half days of lectures and exercises. The rest of time is devoted to the personal project with the hand-in deadline placed on the last day of the fourth week as shown in figure 2. A reasonably short deadline avoids that the personal projects become disproportionately ambitious and take the participants too much time.

	Monday	Tuesday	Wednesday	Thursday	Friday
Week 3	Afternoon	Afternoon	Afternoon	no course	no course
Week 4	no course	no course	no course	no course	p. deadline

Figure 2: Schedule for advanced part

6 Daily schedule of basic part

The teaching consists of lectures and computer exercises which are intertwined. Each day, the session runs over four and a half hours with three coffee breaks included. We want to make sure that the level of engagement is appropriate and that participants are stimulated and challenged while still comfortable with what they are learning.

6.1 Day 1

Interactive lecture

- Examples of why programming is absolutely essential for your life as a scientist. From simple to complex examples. Reproducible science means reproducible programs, some journals require you to publish code.
- Why did we select python as a language (mention examples with competition). But also what is python (imperative and not functional, duck-typed, high-level). Why did we choose version 2.7.x.
- *Theory*: The fundamentals of a modern computer: processing power in terms of flops, basic number representation, the effect of the cascading memory speeds from hard drive to L1 cache.
- A brief introduction to bash: a simple interface to interact with your operating system and file system, and why it should be avoided.
- *Theory*: The three realms to being a good programmer in any language: data, instructions and syntax.

- The two modes we will be using python in: the interactive prompt for live exploring against files for storing finished code as well as code that we are working on.
- Review of the built-in types: `int`, `float`, `str`, `list`, `tuple`, `dict`, `set`.
- Review of control flows: `if`, `else`, `for`, `while`
- Defining functions.

Exercises

- Figure out how long the computer waits in terms of clock cycles when requesting a resource (i) form a local 7200rpm and 10ms seek time hard drive (ii) on a server in California.
- Reflect upon the syntax of a programming language: should it be close to English or not ?
- Simple exercise that involves casting different types and avoiding `TypeError` exceptions. (just a level above the codeacademy ones).
- Simple exercise that involves using control flows (just a level above the codeacademy ones).
- Exercise that makes use of several functions.
- The participants give us feedback on the course.

6.2 Day 2

Interactive lecture

- *Theory*: A brief introduction to algorithms: why and when does the order of computation really matter (simple example when comparing elements to each other = N^2, and example of De Bruijn's graph).
- Mutable and immutable types. Pass by copy or pass by reference.
- List comprehensions.
- File I/O and text manipulation. What is parsing and serializing. Unicode vs ASCII.
- The built-in modules: `os`, `sys`, `re`, etc.
- Regular expressions.

Exercises

- An exercise showing why code duplication is bad.
- An exercise showing reference passing.
- A few exercises on text manipulation and file I/O. Experiment with encodings.
- An exercise using regex patterns.

6.3 Day 3

Interactive lecture

- Linear versus structured programming: advantages and techniques.
- Object oriented programming as the dominating paradigm since it took over C's `structs`.
- How objects link to data structures (DS). How a problem is often solved by simply identifying the right DS.
- Using python packages: how to download and install most of them.
- Packages studied in more depth:
  1. `matplotlib`
  2. `statsmodels`
  3. `pandas`

Exercises

- Example that uses objects and show basic inheritance and basic composition.
- Example that shows the right selection of data structures.
- Example that combines all the three packages: load a dataset with pandas, run a stat model on it, then plot a visualization using matplotlib.

6.4 Day 4

Interactive lecture

- Control your code and share it: introducing `git` and subsequently github.
- From now on, assignments are to be uploaded on github (use the GUI preferably at first).
- Make your own modules that can be imported.
- Introducing PEP8 and coding style guides.
- Comments and docstrings.

Exercises

- Make your account on github and start making simple commits. Use either command line or GUI.
- Given a TSV file that has a table where columns are identifiers and rows are different samples. Also given an index where each identifier is associated with a classification. Some identifiers might belong to the same classification ! Instruction: make a new table linking all different classes to different samples and save it in CSV-UTF8 format.
- Write a simple parser (e.g. FASTA) that responds to certain requirements and passes certain given tests.
- Participants comment on each other's code and try to establish a personal coding style guide.

6.5 Day 5

Interactive lecture

- The different licenses you can put on your code: MIT, BSD, GPL, LGPL, Creative Commons.
- *Theory*: Introduction to the Zen of python and some philosophical remarks.
- How test driven development is the only safe form of development, especially in science where trust is so important. Shocking example: e.g. blog post in QIIME.
- How python can be used to do functional programming. Iterators and generators.

Exercises

- Example showing the advantage of lazy processing data, such as an iterative dataset that can't hold in RAM.
- Write a program that identifies where proteins are hiding in nucleotide sequences (three difficulty steps: simple visual print out, auto frame detection, multi-protein).
- Given a class structure where methods are left blank, automate the processing of N samples through a small series of tools.
- Add automated test to past exercises or other objects.
- Second round of evaluation.

6.6 Day 6

Interactive lecture

- Things that the participants said were not clear enough. This lecture can be designed according to the feedback and the questions we will have gotten. It is a buffer zone of sorts and in any case can be just used for a Q&A session.

Questionnaires

- Evaluation of participant's theoretical knowledge. Questions on the lectures that were given. Written questionnaire, no computer. One hour.
- Practical knowledge assessment. Exercise like the ones they have been doing in the past two weeks. Computer and internet available. Two hours.

7 Daily schedule of advanced part

Here, the participants will develop their own project: a specific analysis he/she wants to program or automate, a tool she/he wants to create to process some data, or even a research-project on programming. The participants are free to choose the direction of their project, but it must be doable in no more than about four days of programming.

7.7 Day 7

Interactive lecture

- Example of how to turn a function oriented script into an object-oriented script and make it better, more readable, more maintainable, and more powerful.
- Nice uses for decorators.
- SQLite3 databases or HDF5 storage for different types of data.
- Premature optimization is the root of all evil. But when you do optimize, run a profiler. Examples of profilers.
- Following the optimization theme: what is garbage collection and how it helps you.

Exercises

- Without googling them, try to come up with one example for each of the points of the "Zen of Python".
- Example where the participant must identify the slow parts of a program.

7.8 Day 8

Interactive lecture

- A few examples of the dark magic that python can do and why it should be avoided. Introspection such as touching `self.__dict__` or even `__module__`, the few cases where `__new__` is justified, metaclasses, monkey patching. Readability is important, don't use black magic.
- *Theory*: The slightly different pythons: CPython, RPython, PyPy, IronPython, Jython, StacklessPython, etc
- *Theory*: The more different pythons: Python 2 and Python 3. What are the usage stats ? Why should I care ?
- Python and speed, when it can actually be useful to write a loop in C.
- Python and concurrency: why it is awful and a brief mention of the Global Interpreter Lock.
- Making user-friendly command-line utilities in python.

Exercises

- Deciphering a script with too-much black magic.
- Small exercise with a few command-line parameters.
- Teachers offer help and guide the participants in defining the skeleton of their project.
- Participants discuss their personal project draft with other participants and try to finalize the blueprint of their course project.

7.9 Day 9

Interactive lecture

- Making pipeline or workflows in python. Libraries or roll your own.
- Examples of previous projects and pipelines. Good and bad sides of design. Stories form the past.
- More complete ways of documenting your code. UML diagrams, sphinx autogeneration, readthedocs.org, github's issues and wiki.
- Methods for collaborating on code and team management. Concepts such as Agile, Sprints, Scrum and Extreme.

Exercises

- Each teacher takes on 5 participants and three groups are created.
- Within groups, short 5 minute presentations for each participant on their final project blueprint.
- The scope and draft of each personal project is validated by the teachers and participants have until the Friday of the next week to complete them. They may come to us in between and each one has a certain amount of meeting time they can use live or via Skype.

8 Reading material

No specific book is associated to this course, and the participants will receive different written materials coming from various sources. However, if the participants want to have such a book to follow and help them, they can use this free one:

http://openbookproject.net/thinkcs/python/english2e/index.html

If this book is read by the participant, it is our opinion that the chapters after number 17 are less interesting in the aspect of python programming for the scientist.

9 Contacts

9.1 Lucas Sinclair

E-mail : <lucas@sinclair.bio>

Title : PhD

Showcase : https://github.com/xapple

Statement : I have always been interested in programming and started at a relatively young age by making small video games. This has helped me in my under-graduate training as an engineer in life sciences and enabled me to easily specialize in bioinformatics. As I have joined a more "classical" wet-lab and field biology department, today, I feel that my programming skills are one of the most valuable skills I could transfer to my peers. Overall, it's a subject I feel very comfortable in and enjoy teaching.

Experience : Veteran python user, taught several python courses already, followed the Academic Teacher Training course, contributed to the open-source python projects.