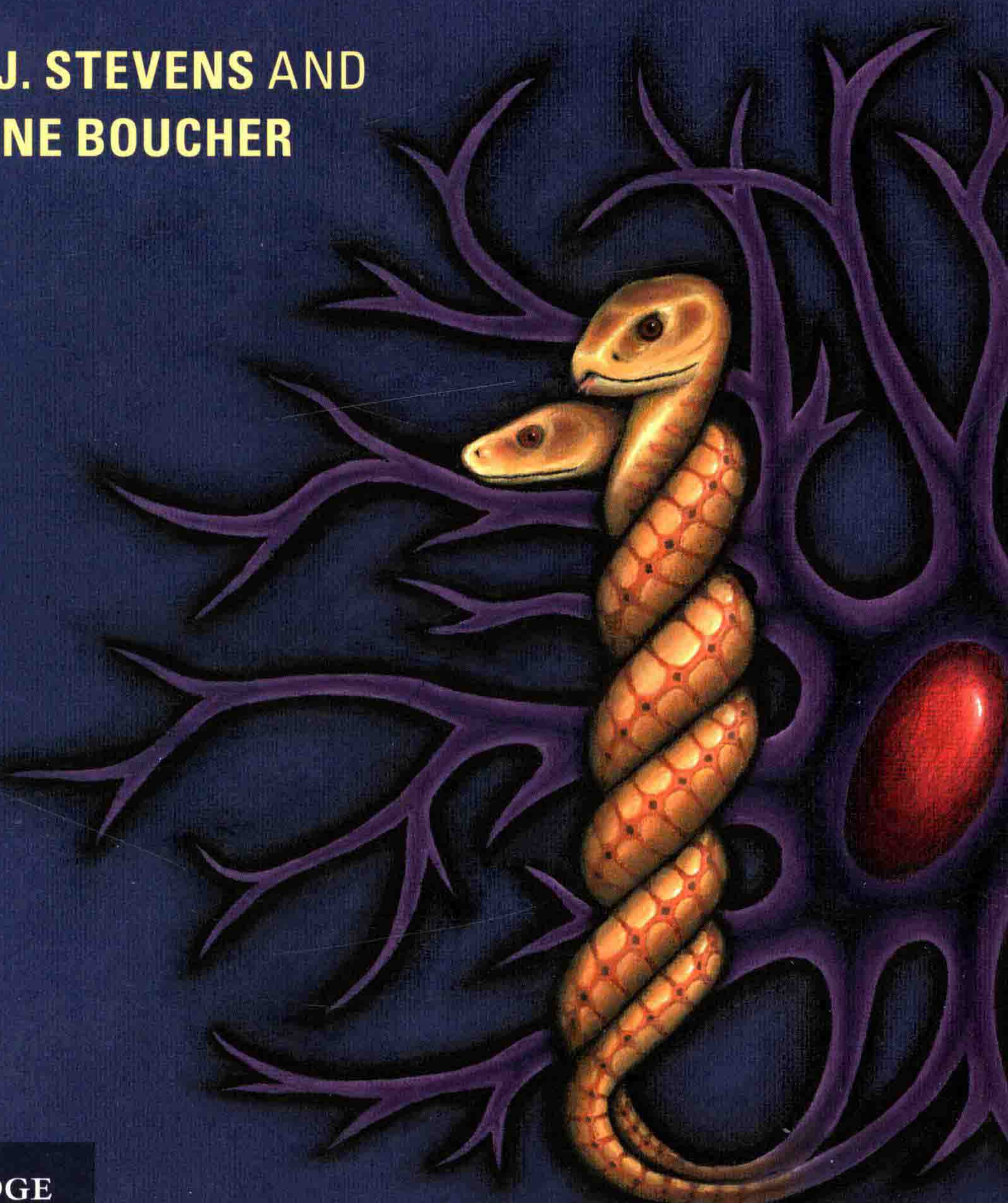
Python Programming for Biology

Bioinformatics and Beyond

TIM J. STEVENS AND
WAYNE BOUCHER



CAMBRIDGE

Python Programming for Biology

Bioinformatics and Beyond

TIM J. STEVENS
MRC Laboratory of Molecular Biology

WAYNE BOUCHER
University of Cambridge

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9780521895835

First published 2015

Printed in the United Kingdom by TJ International Ltd, Padstow Cornwall

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Stevens, Tim J., 1976–
Python programming for biology, bioinformatics, and beyond / Tim J. Stevens, University of Cambridge, Wayne Boucher, University of Cambridge.
pages cm
Includes index.
ISBN 978-0-521-89583-5 (Hardback) – ISBN 978-0-521-72009-0 (Paperback)
1. Biology–Data processing. 2. Python (Computer program language) I. Boucher, Wayne. II. Title.
QH324.2.S727 2014
570.285–dc23 2014021017

ISBN 978-0-521-89583-5 Hardback
ISBN 978-0-521-72009-0 Paperback

Additional resources for this publication at www.cambridge.org/pythonforbiology

Python Programming for Biology

Bioinformatics and Beyond

Do you have a biological question that could be readily answered by computational techniques, but little experience in programming? Do you want to learn more about the core techniques used in computational biology and bioinformatics? Written in an accessible style, this guide provides a foundation for both newcomers to computer programming and those who want to learn more about computational biology. The chapters guide the reader through: a complete beginners' course to programming in Python, with an introduction to computing jargon; descriptions of core bioinformatics methods with working Python examples; scientific computing techniques, including image analysis, statistics and machine learning. This book also functions as a language reference written in straightforward English, covering the most common Python language elements and a glossary of computing and biological terms. This title will teach undergraduates, postgraduates and professionals working in the life sciences how to program with Python, a powerful, flexible and easy-to-use language.

Tim J. Stevens, a biochemist by training, is a Senior Investigator Scientist at the MRC Laboratory of Molecular Biology in Cambridge. He researches three-dimensional genome architecture and provides computational biology oversight, development and training within the Cell Biology Division.

Wayne Boucher, a mathematician and theoretical physicist by training, is a Senior Post-Doctoral Associate and computing technician for the Department of Biochemistry at the University of Cambridge. He teaches undergraduate mathematics and postgraduate programming courses. Wayne is currently developing software for the analysis of biological molecules by nuclear magnetic resonance spectroscopy.

Preface

Many years ago we started programming in Python because we were working on a large computational biology project. In those days choosing Python was not nearly as common as it is today. Nonetheless things worked out well, and as our expertise grew it seemed only natural that we should run some elementary Python courses for the School of Biology at the University of Cambridge, where we were employed. The basis for those courses is what turned into the initial idea for this book. While there were many books about getting started with Python and some that were tailored to bioinformatics, we felt that there was still some room for what we wanted to put across. We began with the idea that we could write some chapters in relatively straightforward English that were aimed at biologists, who might be complete novices at programming, and have other sections that are useful to a more experienced programmer. Also, given that we didn't consider ourselves to be typical bioinformaticians, we were thinking more broadly than just sequence-based informatics, though naturally such things would be included. We felt that although we couldn't anticipate all the requirements of a biological programmer there were nonetheless a number of key concepts and techniques which we could try to explain. The end result is hopefully a toolkit of ideas and examples which can be applied by biologists in a variety of situations.

Tim J. Stevens and
Wayne Boucher,
Cambridge, January 2014

Acknowledgements

We extend our sincere thanks to a group of intrepid volunteers who have been invaluable in the proof-reading and testing of this book: Olga Tkachenko, Magnus Lundborg, Neil Rzechorzek, Rasmus Fogh, Simon Fraser and Tom Drury.

Special thanks also go to David Judge, who has run the bioinformatics teaching facility at Cambridge for many years and who made it very easy to give the Python courses that eventually led to this book.

We acknowledge the support of the Medical Research Council and the Biotechnology and Biological Sciences Research Council, the UK funding bodies who have funded the scientific projects that we have been involved with over the years. This has allowed us to use and develop our Python programming skills while remaining gainfully employed.

Contents

1 Prologue

Contents

Python programming for biology

One of the main aims of this book is to empower the average researcher in the life sciences, who may have a pertinent scientific question that can be readily answered by computational techniques, but who doesn't have much, if any, experience with programming. For many in this position, the task of writing a program in a computer language is a bottleneck, if not an impassable barrier. Often, the task is daunting and seems to require a significant investment of time. The task is also subject to the barriers presented by a vocabulary filled with jargon and a seemingly steep learning curve for those people who were not trained in computing or have no inclination to become computer specialists. With this in mind for the novice programmer, one ought to start with the language that is the easiest to get to grips with, and at the time of writing we believe that that language is Python. This is not to say that we have made a compromise by choosing a language that is easy to learn but which is not powerful or fully featured. Python is certainly a very rich and capable way of programming, even for very large projects; otherwise we authors wouldn't be using it for our own scientific work.

A second main aim of this book is to use Python as a means to illustrate some of what is going on within biological computing. We hope our explanations will show you the scientific context of why something is done with computers, even if you are a newcomer to biology or medical sciences. Even where a popular biological program is not written in Python, or if you are a programmer who has good reason for using another language, we can still use Python as a way of illustrating the major principles of programming for biology. We feel that many of the most useful biological programs are based on combinations of simple principles that almost anyone can understand. By trying to separate the core concepts from the obfuscation and special cases, we aim to provide an overview of techniques and strategies that you can use as a resource in your own research. Virtually all of the examples in this book are working code that can be run and are based on real problems or programs within biological computing. The examples can then be adapted, altered and combined to enable you to program whatever you need.

We wish to make clear that this book intends to show you what sort of things can be done and how to begin. It does not intend to offer a deep and detailed analysis of specific biological and computational problems. This is not a typical scientific book, given that we don't always go for the most detailed or up-to-date examples. Given the choice, we aim to give a broad-based understanding to newcomers and avoid what some may consider pedantry.

No doubt some people will think our approach somewhat too simplistic, but if you know enough to know the difference then we don't recommend looking to this book for those kinds of answers. Likewise, there is only room for so many examples and we cannot cover all of the scientific methods (including Python software libraries) that we would want to. Hopefully though, we give the reader enough pointers to make a good start.

Choosing Python

It is perhaps important to include a short justification to say why we have written this book for the Python programming language; after all, we can choose from several alternative languages. Certainly Python is the language that we the authors write in on a daily basis, but this familiarity was actually born out of a conscious decision to use Python for a large biological programming project after having tried and considered a number of popular alternatives. Aside from Python, the languages that we have commonly come across in today's biological community include: C, C++, FORTRAN, Java, Matlab, Perl, R and Ruby. Specific comparison with some of these languages will be made at various points in the book, but there are some characteristics of Python that we enjoy, which we feel would not be available to the same level or in the same combination in any other language.

We like the clear and consistent layout that directs the programmer away from obfuscated program code and towards an elegantly readable solution; this becomes especially important when trying to work out what someone else's program does, or even what your own material does several years later. We like the way that Python has object orientation at its heart, so you can use this powerful way to organise your data while still having the easy look and feel of Python. This also means that by learning the language basics you automatically become familiar with the very useful object-oriented approach. We like that Python generally requires fewer lines of program code than other languages to do the equivalent job, and that it often seems so much less tedious to write.

It is important to make it clear that we would not currently use Python for every programming task in the life sciences. Python is not a perfect language. As it stands currently for some specialised tasks, particularly those that require fast mathematical calculations which are not supported by the numeric Python modules, we actively promote working with a Python extension such as Cython, or some faster alternative language. However, we heartily recommend that Python be used to administer the bookkeeping while the faster alternative provides extra modules that act as a fast calculation engine. To this end, in Chapter 27 we will show you how you can seamlessly mesh the Python language with Cython and also with the compiled language C, to give all the benefits of Python and very fast calculations.

Python's history and versions

The Python[1] programming language was the creation of Guído van Rossum. It is because of his innovation and continuing support that Python is popular and continues to grow. The Python programming community has afforded Guido the honour of the title 'benevolent

[1] The name itself derives from Monty Python, which is why you'll find the occasional honorary reference to 'spam', 'dead parrot' etc. when arbitrary examples are given.

dictator for life'. What this means is that despite the fact that many aspects of Python are developed by a large community, Guido has the ultimate say in what goes into Python. Although not bound in any legality, everyone abides by Guido's decisions, even if at times some people are surprised by what he decides. We believe that this situation has largely benefited Python by ensuring that the philosophy remains unsullied. Seemingly often, a committee decision has the tendency to try to appease all views and can become tediously slow with indecision; too timid to make any bold, yet improving moves. The Python programming community has a large role in criticising Python and guiding its future development, but when a decision needs to be made, it is one that everyone accepts. Certainly there could be a big disagreement in the future, but so far the benevolent dictator's decisions have always taken the community with him.

There have been several, and in our opinion improving, versions of the Python programming language. All versions before Python 3 share a very high degree of backward-compatibility, so that code written for version 1.5 will still (mostly) work with say version 2.7 with few problems. Python 3 is not as compatible with older versions, but this seems a reasonable price to be paid to keep things moving forward and eradicate some of the undesired legacy that earlier versions have built up. Rest assured though, version 3 remains similar enough in look and feel to the older Pythons, even if it is not exactly the same, and the examples in this book work with both Python 2 and Python 3 except where specifically noted. Also, included with the release of Python 3 is a conversion program '2to3' which will attempt to automatically change the relevant parts of a version 2 program so that it works with version 3. This will not be able to deal with every situation, but it will handle the vast majority and save considerable effort.

For this book we will assume that you are using Python version 2.6 or 2.7 or 3. Some bits, however, that use some newer features will not work with versions prior to 2.6 without alteration. We feel that it is better to use the best available version, rather than write in a deliberately archaic manner, which would detract from clarity.

Bioinformatics

The field of bioinformatics has emerged as we have discovered, through experimentation, large amounts of DNA and protein sequence information. In its most conservative sense bioinformatics is the discipline of extracting scientific information by the study of these biological sequences, which, because of the large amount of data, must be analysed by computer. Initially this encompassed what most biological computing was about, but we contend that this was simply where biomolecular computing began and that it has far to go. The informatics of biological systems these days includes the study of molecular structures, including their dynamics and interactions, enzymatic activity, medical and pharmacological statistics, metabolic profiles, system-wide modelling and the organisation of experimental procedures, to name only a subset. It is within this wider context that this book is placed.

At present the programming language that is historically most famous for being used with bioinformatics is probably Perl, which is notable for its ability to manipulate sequences, particularly when stored as letters within formatted text. It also has a library of modules available to perform many common bioinformatics tasks, collectively named BioPerl. In this arena Python can do everything that Perl can. There is a Python equivalent of BioPerl, unsurprisingly named BioPython, and at this time the uptake of Python within the bioinformatics community is growing, which is not surprising, given our belief that it is an easier but more powerful language to work with. It is important to note that although some of the BioPython modules will certainly be discussed in the course of this book (and we would generally advise using tested, existing code wherever possible to make your programs easier to write and understand) the explanations and examples will be more to do with understanding what is going on underneath. We aim to avoid this book simply becoming a brochure for existing programs where you don't have to know the inner workings.

Computer platforms and installations

Python is available for every commonly used computer operating system including versions of Microsoft Windows, Mac OS X, Linux and UNIX. With Windows you will generally have to download and install Python, as it is not included as standard. On most new Mac OS X, Linux and UNIX systems Python is included as standard (indeed some parts of Linux operating systems are themselves written with Python), although you should check to see which version of Python you have: typing '`python`' at a command line reveals the version. For a list of website locations where you can download Python for various platforms see the reference section at the end of this book or the Cambridge University Press site: http://www.cambridge.org/pythonforbiology.

Precisely because Python is available for and can be run on many different computer platforms, any programs you write will generally be able to be run on all computer systems. However, there are a few important caveats you should be aware of. Although Python as a language is interpreted in the same way on every computer system, when it comes to interacting with the operating system (Windows, Mac OS X, Linux ...), things can work differently on different computers. This is a problem that all cross-platform computing languages face. You will probably come across this in your Python programs when dealing with files and the directories that contain them. Although each operating system will have its own nuances, once you are aware of the differences it is a relatively simple job to ensure that your programs work just as well under any common operating system, and we will cover details of this as required in the subsequent chapters.

2 A beginners' guide

Contents

Programming principles

The Python language can be viewed as a formalised system of understanding instructions (represented by letters, numbers and other funny characters) and acting upon those directions. Quite naturally, you have to put something in to get something out, and what you are going to be passing to Python is a series of commands. Python is itself a computer program, which is designed to interpret commands that are written in the Python language, and then act by executing what these instructions direct. A programmer will sometimes refer to such commands collectively as 'code'.

Interpreting commands

So, to our first practical point; to get the Python interpreter to do something we will give it some commands in the form of a specially created piece of text. It is possible to give Python a series of commands one at a time, as we slowly type something into our computer. However, while giving Python instructions line by line is useful if you want to test out something small, like the examples in this chapter, for the most part this method of issuing commands is impractical. What we usually do instead is create all of the lines of text representing all the instructions, written as commands in the Python language, and store the whole lot in a file. We can then activate the Python interpreter program so that it reads all of the text from the file and acts on all of the commands issued within. A series of commands that we store together in such a way, and which do a specific job, can be considered as a computer program.[1] If you would like to try any of the examples given in the book the next chapter will tell you how to actually get started. The initial intention, however, is mostly to give you a flavour of Python and introduce a few key principles.

```
mass = 5.9736
volume = 1.08321
density = mass/volume
print(density)
```

An example of a very simple, four-line Python program that performs a calculation and displays the result.

Reusable functionality

When writing programs in the Python language, which the Python interpreter can then use, we are not restricted to reading commands from only one file. It is a very common practice to have a program distributed over a number of different files. This helps to organise writing of the program, as you can put different specialised parts of your instructions into different files that you can develop separately, without having to wade through large amounts of text. Also, and perhaps most importantly, having Python commands in multiple files enables different programs to share a set of commands. With shared files, the distinction between which commands belong to one program and which belong to another is mostly meaningless. As such, we typically refer to such a shared file as a *module*.

In Python you will use modules on a regular basis. And, as you might have already guessed, the idea is to have modules containing a series of commands which perform a function that would be useful for several programs, perhaps in quite different situations. For example, you could write a module which contains the commands to do a statistical analysis on some numeric data. This would be useful to any program that needs to run that kind of analysis, as hopefully we have written the statistics module in such a way that the precise amount and source of the numeric data that we send to the module is irrelevant. Whenever we use a module we are avoiding having to write new Python commands, and are hopefully using something that has been tried and tested and is known to work.

```
from Alignments import sequenceAlign
sequence1 = 'GATTACAGC'
sequence2 = 'GTATTAAT'
print(sequenceAlign(sequence1, sequence2))
```

A Python example where general functionality, to align two sequences of letters, is imported from a module called `Alignments`, *which was defined elsewhere.*

When working with Python there is already a long list of pre-made modules that you can use. For example, there are modules to perform common mathematical operations, to interact with

[1] Not 'programme', even in the UK.

the operating system and to search for patterns of symbols within text. These are all generally very useful, and as such they are included as standard whenever you have Python installed. You will still have to load, or *import*, these modules into a program to use them, but in essence you can think of these modules as a convenient way of extending the vocabulary of the Python language when you need to. By the same token, you don't have to load any modules that are not going to be useful, which might slow things down or use unnecessary computer memory.

Types of data

Before going on to give a more detailed tutorial we will first describe a little about the construction and makeup of commands written in the Python language. Writing the command code for a program involves thinking about items of data. There can be many different kinds of data, from different origins, that we would wish to manipulate with a computer. Typically we will represent the smallest units of this information as numbers or text. We can organise such numbers and text into structured arrangements, for example, to create a list of data, and we can then manipulate this entire larger container, with all of its underlying elements, as a single unit. For example, given a list containing numbers you could extract the first number from the list, or maybe get the list in reverse order.

```
numbers = [6, 0, 2, 2, 1, 4, 1, 5]
numbers.reverse()

print(numbers)
```

Defining a list of numbers as a single entity and then reversing its order, before printing the result to the screen.

In Python, as in many languages, there are some standard types of data-containing structures that form the basis of most programs, and which are very easy to create and fill with information. But you are not limited to these standard data structures; you can create your own data organisation. For example, you could create a data structure called a `Person`, which can store the name, sex, height and age of real people. In a program, just as you could get the first element of data stored in a list, so too could you extract the number that represents the age of a `Person` data structure. Going further, you could create many `Person` data structures and organise them further by placing them into lists. A data structure can appear inside the organisation of many other data structures, so a single `Person` could appear in several different lists (for example, organised by age, sex or whatever) or a `Person` could contain references to other `Person` data structures to indicate the relationships between parents and children.

Python objects

This is where we can introduce the concept of an *object*. The `Person` data structure described above would commonly be referred to as a `Person` object. Indeed, all of the organised data structures in Python, including the simple inbuilt ones, are referred to as Python objects. So numbers, text and lists are all kinds of objects. Not every programming language formalises things in this way, but it will start to feel natural once you are used to Python, and means that the form of the programming language is the same whatever type of object is being manipulated.