PLOS | COMPUTATIONAL BIOLOGY

EDUCATION

An Introduction to Programming for Bioscientists: A Python-Based Primer

Berk Ekmekci[☯], **Charles E. McAnany**[☯], **Cameron Mura***

Department of Chemistry, University of Virginia, Charlottesville, Virginia, United States of America

☯ These authors contributed equally to this work.
* cmura@muralab.org

Abstract

Computing has revolutionized the biological sciences over the past several decades, such that virtually all contemporary research in molecular biology, biochemistry, and other biosciences utilizes computer programs. The computational advances have come on many fronts, spurred by fundamental developments in hardware, software, and algorithms. These advances have influenced, and even engendered, a phenomenal array of bioscience fields, including molecular evolution and bioinformatics; genome-, proteome-, transcriptome- and metabolome-wide experimental studies; structural genomics; and atomistic simulations of cellular-scale molecular assemblies as large as ribosomes and intact viruses. In short, much of post-genomic biology is increasingly becoming a form of computational biology. The ability to design and write computer programs is among the most indispensable skills that a modern researcher can cultivate. Python has become a popular programming language in the biosciences, largely because (i) its straightforward semantics and clean syntax make it a readily accessible first language; (ii) it is expressive and well-suited to object-oriented programming, as well as other modern paradigms; and (iii) the many available libraries and third-party toolkits extend the functionality of the core language into virtually every biological domain (sequence and structure analyses, phylogenomics, workflow management systems, etc.). This primer offers a basic introduction to coding, via Python, and it includes concrete examples and exercises to illustrate the language's usage and capabilities; the main text culminates with a final project in structural bioinformatics. A suite of Supplemental Chapters is also provided. Starting with basic concepts, such as that of a "variable," the Chapters methodically advance the reader to the point of writing a graphical user interface to compute the Hamming distance between two DNA sequences.

CrossMark
click for updates

OPEN ACCESS

Citation: Ekmekci B, McAnany CE, Mura C (2016) An Introduction to Programming for Bioscientists: A Python-Based Primer. PLoS Comput Biol 12(6): e1004867. doi:10.1371/journal.pcbi.1004867

Editor: Francis Ouellette, Ontario Institute for Cancer Research, CANADA

Published: June 7, 2016

Copyright: © 2016 Ekmekci et al. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Funding: Portions of this work were supported by the University of Virginia, the Jeffress Memorial Trust (J-971), a UVa Harrison undergraduate research award (BE), NSF grant DUE-1044858 (CM), and NSF CAREER award MCB-1350957 (CM). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

Author Summary

Contemporary biology has largely become computational biology, whether it involves applying physical principles to simulate the motion of each atom in a piece of DNA, or using machine learning algorithms to integrate and mine "omics" data across whole cells (or even entire ecosystems). The ability to design algorithms and program computers, even at a novice level, may be the most indispensable skill that a modern researcher can

Takeda EX1123

cultivate. As with human languages, computational fluency is developed actively, not passively. This self-contained text, structured as a hybrid primer/tutorial, introduces any biologist—from college freshman to established senior scientist—to basic computing principles (control-flow, recursion, regular expressions, etc.) and the practicalities of programming and software design. We use the Python language because it now pervades virtually every domain of the biosciences, from sequence-based bioinformatics and molecular evolution to phylogenomics, systems biology, structural biology, and beyond. To introduce both coding (in general) and Python (in particular), we guide the reader via concrete examples and exercises. We also supply, as Supplemental Chapters, a few thousand lines of heavily-annotated, freely distributed source code for personal study.

Introduction

Motivation: Big Data and Biology

Datasets of unprecedented volume and heterogeneity are becoming the norm in science, and particularly in the biosciences. High-throughput experimental methodologies in genomics [1], proteomics [2], transcriptomics [3], metabolomics [4], and other "omics" [5–7] routinely yield vast stores of data on a system-wide scale. Growth in the quantity of data has been matched by an increase in heterogeneity: there is now great variability in the types of relevant data, including nucleic acid and protein sequences from large-scale sequencing projects, proteomic data and molecular interaction maps from microarray and chip experiments on entire organisms (and even ecosystems [8–10]), three-dimensional (3D) coordinate data from international structural genomics initiatives, petabytes of trajectory data from large-scale biomolecular simulations, and so on. In each of these areas, volumes of raw data are being generated at rates that dwarf the scale and exceed the scope of conventional data-processing and data-mining approaches.

The intense data-analysis needs of modern research projects feature at least three facets: data *production*, *reduction/processing*, and *integration*. Data production is largely driven by engineering and technological advances, such as commodity equipment for next-gen DNA sequencing [11–13] and robotics for structural genomics [14,15]. Data reduction requires efficient computational processing approaches, and data integration demands robust tools that can flexibly represent data (*abstractions*) so as to enable the detection of correlations and interdependencies (via, e.g., machine learning [16]). These facets are closely coupled: the rate at which raw data is now produced, e.g., in computing molecular dynamics (MD) trajectories [17], dictates the data storage, processing, and analysis needs. As a concrete example, the latest generation of highly-scalable, parallel MD codes can generate data more rapidly than they can be transferred via typical computer network backbones to local workstations for processing. Such demands have spurred the development of tools for "on-the-fly" trajectory analysis (e.g., [18,19]) as well as generic software toolkits for constructing parallel and distributed data-processing pipelines (e.g., [20] and S2 Text, §2). To appreciate the scale of the problem, note that calculation of all-atom MD trajectories over biologically-relevant timescales easily leads into petabyte-scale computing. Consider, for instance, a biomolecular simulation system of modest size, such as a 100-residue globular protein embedded in explicit water (corresponding to $\approx 10^5$ particles), and with typical simulation parameters (32-bit precision, atomic coordinates written to disk, in binary format, for every ps of simulation time, etc.). Extending such a simulation to 10 μs duration—which may be at the low end of what is deemed biologically relevant for the system—would give an approximately 12-terabyte trajectory ($\approx 10^5$particles × 3 coordinates/particle/frame × 10^7 frames × 4 bytes/coordinate = 12TB). To validate or otherwise follow-up

predictions from a single trajectory, one might like to perform an additional suite of >10 such simulations, thus rapidly approaching the peta-scale.

Scenarios similar to the above example occur in other biological domains, too, at length-scales ranging from atomic to organismal. Atomistic MD simulations were mentioned above. At the molecular level of individual genes/proteins, an early step in characterizing a protein's function and evolution might be to use sequence analysis methods to compare the protein sequence to every other known sequence, of which there are tens of millions [21]. Any form of 3D structural analysis will almost certainly involve the Protein Data Bank (PDB; [22]), which currently holds over 10^5 entries. At the cellular level, proteomics, transcriptomics, and various other "omics" areas (mentioned above) have been inextricably linked to high-throughput, big-data science since the inception of each of those fields. In genomics, the early bottleneck—DNA sequencing and raw data collection—was eventually supplanted by the problem of processing raw sequence data into derived (secondary) formats, from which point meaningful conclusions can be gleaned [23]. Enabled by the amount of data that can be rapidly generated, typical "omics" questions have become more subtle. For instance, simply assessing sequence similarity and conducting functional annotation of the open reading frames (ORFs) in a newly sequenced genome is no longer the end-goal; rather, one might now seek to derive networks of biomolecular functions from sparse, multi-dimensional datasets [24]. At the level of tissue systems, the modeling and simulation of inter-neuronal connections has developed into a new field of "connectomics" [25,26]. Finally, at the organismal and clinical level, the promise of personalized therapeutics hinges on the ability to analyze large, heterogeneous collections of data (e.g., [27]). As illustrated by these examples, all bioscientists would benefit from a basic understanding of the computational tools that are used daily to collect, process, represent, statistically manipulate, and otherwise analyze data. In every data-driven project, the overriding goal is to transform raw data into new biological principles and knowledge.

A New Kind of Scientist

Generating knowledge from large datasets is now recognized as a central challenge in science [28]. To succeed, each type of aforementioned data-analysis task hinges upon three things: greater computing power, improved computational methods, and computationally fluent scientists. Computing power is only marginally an issue: it lies outside the scope of most biological research projects, and the problem is often addressed by money and the acquisition of new hardware. In contrast, computational methods—improved algorithms, and the software engineering to implement the algorithms in high-quality codebases—are perpetual goals. To address the challenges, a new era of scientific training is required [29–32]. There is a dire need for biologists who can collect, structure, process/reduce, and analyze (both numerically and visually) large-scale datasets. The problems are more fundamental than, say, simply converting data files from one format to another ("data-wrangling"). Fortunately, the basics of the necessary computational techniques can be learned quickly. Two key pillars of computational fluency are (i) a working knowledge of some programming language and (ii) comprehension of core computer science principles (data structures, sort methods, etc.). All programming projects build upon the same set of basic principles, so a seemingly crude grasp of programming essentials will often suffice for one to understand the workings of very complex code; one can develop familiarity with more advanced topics (graph algorithms, computational geometry, numerical methods, etc.) as the need arises for particular research questions. Ideally, computational skills will begin to be developed during early scientific training. Recent educational studies have exposed the gap in life sciences and computer science knowledge among young scientists, and interdisciplinary education appears to be effective in helping bridge the gap [33,34].

Programming as the Way Forward

For many of the questions that arise in research, software tools have been designed. Some of these tools follow the Unix tradition to "make each program do one thing well" [35], while other programs have evolved into colossal applications that provide numerous sophisticated features, at the cost of accessibility and reliability. A small software tool that is designed to perform a simple task will, at some point, lack a feature that is necessary to analyze a particular type of dataset. A large program may provide the missing feature, but the program may be so complex that the user cannot readily master it, and the codebase may have become so unwieldy that it cannot be adapted to new projects without weeks of study. Guy Steele, a highly-regarded computer scientist, noted this principle in a lecture on programming language design [36]:

> "*I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time.*"

Programming languages provide just such a tool. Instead of supplying every conceivable feature, languages provide a small set of well-designed features and powerful tools to compose these features in new ways, using logical principles. Programming allows one to control every aspect of data analysis, and libraries provide commonly-used functionality and pre-made tools that the scientist can use for most tasks. A good library provides a simple interface for the user to perform routine tasks, but also allows the user to tweak and customize the behavior in any way desired (such code is said to be *extensible*). The ability to compose programs into other programs is particularly valuable to the scientist. One program may be written to perform a particular statistical analysis, and another program may read in a data file from an experiment and then use the first program to perform the analysis. A third program might select certain datasets—each in its own file—and then call the second program for each chosen data file. In this way, the programs can serve as modules in a computational workflow.

On a related note, many software packages supply an application programming interface (API), which exposes some specific set of functionalities from the codebase without requiring the user/programmer to worry about the low-level implementation details. A well-written API enables users to combine already established codes in a modular fashion, thereby more efficiently creating customized new tools and pipelines for data processing and analysis.

A program that performs a useful task can (and, arguably, should [37]) be distributed to other scientists, who can then integrate it with their own code. Free software licenses facilitate this type of collaboration, and explicitly encourage individuals to enhance and share their programs [38]. This flexibility and ease of collaborating allows scientists to develop software relatively quickly, so they can spend more time integrating and mining, rather than simply processing, their data.

Data-processing workflows and pipelines that are designed for use with one particular program or software environment will eventually be incompatible with other software tools or workflow environments; such approaches are often described as being *brittle*. In contrast, algorithms and programming logic, together with robust and standards-compliant data-exchange formats, provide a completely universal solution that is portable between different tools. Simply stated, any problem that can be solved by a computer can be solved using any programming language [39,40]. The more feature-rich or *high-level* the language, the more concisely can a data-processing task be expressed using that language (the language is said to be *expressive*). Many high-level languages (e.g., Python, Perl) are executed by an *interpreter*, which is a program that reads source code and does what the code says to do. Interpreted languages are not as numerically efficient as

lower-level, compiled languages such as C or Fortran. The source code of a program in a *compiled* language must be converted to machine-specific instructions by a compiler, and those low-level machine code instructions (*binaries*) are executed directly by the hardware. Compiled code typically runs faster than interpreted code, but requires more work to program. High-level languages, such as Python or Perl, are often used to prototype ideas or to quickly combine modular tools (which may be written in a lower-level language) into "scripts"; for this reason they are also known as *scripting languages*. Very large programs often provide a scripting language for the user to run their own programs: Microsoft Office has the VBA scripting language, PyMOL [41] provides a Python interpreter, VMD [42] uses a Tcl interpreter for many tasks, and Coot [43] uses the Scheme language to provide an API to the end-user. The deep integration of high-level languages into packages such as PyMOL and VMD enables one to extend the functionality of these programs via both scripting commands (e.g., see PyMOL examples in [44]) and the creation of semi-standalone plugins (e.g., see the VMD plugin at [45]). While these tools supply interfaces to different programming languages, the fundamental concepts of programming are preserved in each case: a script written for PyMOL can be transliterated to a VMD script, and a closure in a Coot script is roughly equivalent to a closure in a Python script (see Supplemental Chapter 13 in S1 Text). Because the logic underlying computer programming is universal, mastering one language will open the door to learning other languages with relative ease. As another major benefit, the algorithmic thinking involved in writing code to solve a problem will often lead to a deeper and more nuanced understanding of the scientific problem itself.

Why Python? (And Which Python?)

Python is the programming language used in this text because of its clear syntax [40,46], active developer community, free availability, extensive use in scientific communities such as bioinformatics, its role as a scripting language in major software suites, and the many freely available scientific libraries (e.g., BioPython [47]). Two of these characteristics are especially important for our purposes: (i) a clean syntax and straightforward semantics allow the student to focus on core programming concepts without the distraction of difficult syntactic forms, while (ii) the widespread adoption of Python has led to a vast base of scientific libraries and toolkits for more advanced programming projects [20,48]. As noted in the S2 Text (§1), several languages other than Python have also seen widespread use in the biosciences; see, e.g., [46] for a comparative analysis of some of these languages. As described by Hinsen [49], Python's particularly rapid adoption in the sciences can be attributed to its powerful and versatile combination of features, including characteristics intrinsic to the language itself (e.g., expressiveness, a powerful object model) as well as extrinsic features (e.g., community libraries for numerical computing).

Two versions of Python are frequently encountered in scientific programming: Python 2 and Python 3. The differences between these are minor, and while this text uses Python 3 exclusively, most of the code we present will run under both versions of Python. Python 3 is being actively developed and new features are added regularly; Python 2 support continues mainly to serve existing ("legacy") codes. New projects should use Python 3.

Role and Organization of This Text

This work, which has evolved from a modular "Programming for Bioscientists" tutorial series that has been offered at our institution, provides a self-contained, hands-on primer for general-purpose programming in the biosciences. Where possible, explanations are provided for key foundational concepts from computer science; more formal, and comprehensive, treatments can be found in several computer science texts [39,40,50] as well as bioinformatics titles, from both theoretical [16,51] and more practical [52–55] perspectives. Also, this work complements

other practical Python primers [56], guides to getting started in bioinformatics (e.g., [57,58]), and more general educational resources for scientific programming [59].

Programming fundamentals, including variables, expressions, types, functions, and control flow and recursion, are introduced in the first half of the text ("Fundamentals of Programming"). The next major section ("Data Collections: Tuples, Lists, For Loops, and Dictionaries") presents data structures for collections of items (lists, tuples, dictionaries) and more control flow (loops). Classes, methods, and other basics of object-oriented programming (OOP) are described in "Object-Oriented Programming in a Nutshell". File management and input/output (I/O) is covered in "File Management and I/O", and another practical (and fundamental) topic associated with data-processing—regular expressions for string parsing—is covered in "Regular Expressions for String Manipulations". As an advanced topic, the text then describes how to use Python and Tkinter to create graphical user interfaces (GUIs) in "An Advanced Vignette: Creating Graphical User Interfaces with Tkinter". Python's role in general scientific computing is described as a topic for further exploration ("Python in General-Purpose Scientific Computing"), as is the role of software licensing ("Python and Software Licensing") and project management via version control systems ("Managing Large Projects: Version Control Systems"). Exercises and examples occur throughout the text to concretely illustrate the language's usage and capabilities. A final project ("Final Project: A Structural Bioinformatics Problem") involves integrating several lessons from the text in order to address a structural bioinformatics question.

A collection of Supplemental Chapters (S1 Text) is also provided. The Chapters, which contain a few thousand lines of Python code, offer more detailed descriptions of much of the material in the main text. For instance, variables, functions and basic control flow are covered in Chapters 2, 3, and 5, respectively. Some topics are examined at greater depth, taking into account the interdependencies amongst topics—e.g., functions in Chapters 3, 7, and 13; lists, tuples, and other collections in Chapters 8, 9, and 10; OOP in Chapters 15 and 16. Finally, some topics that are either intermediate-level or otherwise not covered in the main text can be found in the Chapters, such as modules in Chapter 4 and lambda expressions in Chapter 13. The contents of the Chapters are summarized in Table 1 and in the S2 Text (§3, "Sample Python Chapters").

Using This Text

This text and the Supplemental Chapters work like the lecture and lab components of a course, and they are designed to be used in tandem. For readers who are new to programming, we suggest reading a section of text, including working through any examples or exercises in that section, and then completing the corresponding Supplemental Chapters before moving on to the next section; such readers should also begin by looking at §3.1 in the S2 Text, which describes how to interact with the Python interpreter, both in the context of a Unix Shell and in an integrated development environment (IDE) such as IDLE. For bioscientists who are somewhat familiar with a programming language (Python or otherwise), we suggest reading this text for background information and to understand the conventions used in the field, followed by a study of the Supplemental Chapters to learn the syntax of Python. For those with a strong programming background, this text will provide useful information about the software and conventions that commonly appear in the biosciences; the Supplemental Chapters will be rather familiar in terms of algorithms and computer science fundamentals, while the biological examples and problems may be new for such readers.

Typographic Conventions

The following typographic conventions appear in the remainder of this text: (i) all computer code is typeset in a `monospace` font; (ii) many terms are defined contextually, and are

Table 1. Overview of the Supplemental Chapters (S1 Text).

Chapter	Name	Topics
00	Setup	Commenting code. Running programs. Testing imports.
01	Introduction	Print function. Elementary arithmetic. Function definition syntax. Strings.
02	Variables	Variables and assignment. Semantics of `x = x + 1`
03	Functions, I	Arguments and returns in functions.
04	Modules	Importing code from existing libraries.
05	Control Flow, I	`if` statements, indentation in Python. `%` and `//` operators. Boolean algebra.
06	Control Flow, II	Iteration using `for` and `while`.
07	Functions, II	Recursion and recursive problem decomposition. Collatz conjecture.
08	Collections, I	Syntax and semantics of sequences. Creation, modification, indexing, and slicing of tuples, lists, strings. Stack and heap storage. String formatting.
09	Collections, II	First-class functions. Techniques of list processing: `filter`, `fold`, `map`.
10	Collections, III	Syntax of dictionaries. Key-value pairs.
11	File I/O	Reading and writing files. Importance of leaving a file in a consistent state.
12	Graphics	Windows and basic graphics. Mouse events. External documentation.
13	Functions, III	Lambdas and functors. Nested function definitions and closures. Currying.
14	Exceptions	Indication of and recovery from errors. Exception hierarchy.
15	Classes, I	Object-oriented programming style; methods, members, inheritance. Assignment semantics.
16	Classes, II	Case studies of complex classes. Iterators.
17	Regexes	Essentials of regular expressions.
18	Tkinter, I	Creation of windows and widgets. Geometry managers.
19	Tkinter, II	Adding functionality to widgets. Tkinter variables.

The Supplemental Chapters (S1 Text) consist of a few thousand lines of Python code, heavily annotated with explanatory material and covering the topics summarized here. In general, the material ranges from relatively basic to more intermediate and advanced levels as the Chapters progress. The latest versions of the Chapters are available at http://p4b.muralab.org.

doi:10.1371/journal.pcbi.1004867.t001

introduced in *italics*; (iii) **boldface** type is used for occasional emphasis; (iv) single (') and double ("") quote marks are used either to indicate colloquial terms or else to demarcate character or word boundaries amidst the surrounding text (for clarity); (v) module names, filenames, pseudocode, and GUI-related strings appear as sans-serif text; and (vi) regular expressions are offset by `a gray background`, e.g. `.` denotes a period. We refer to delimiters in the text as (parentheses), [brackets], and {braces}.

Blocks of code are typeset in `monospace` font, with keywords in bold and strings in italics. Output appears on its own line without a line number, as in the following example:

```
1 if(True):
2   print("hello")
  hello
3 exit(0)
```

Fundamentals of Programming

Variables and Expressions

The concept of a variable offers a natural starting point for programming. A *variable* is a name that can be set to represent, or "hold," a specific value. This definition closely parallels that

found in mathematics. For example, the simple algebraic statement `x = 5` is interpreted mathematically as introducing the variable *x* and assigning it the value 5. When Python encounters that same statement, the interpreter generates a variable named `x` (literally, by allocating memory), and assigns the value `5` to the variable name. The parallels between variables in Python and those in arithmetic continue in the following example, which can be typed at the prompt in any Python shell (§3.1 of the S2 Text describes how to access a Python shell):

```
1 x = 5
2 y = 7
3 z = x + 2 * y
4 print(z)
  19
```

As may be expected, the value of `z` is set equal to the sum of `x` and `2*y`, or in this case 19. The `print()` function makes Python output some text (the *argument*) to the screen; its name is a relic of early computing, when computers communicated with human users via ink-on-paper printouts. Beyond addition (`+`) and multiplication (`*`), Python can perform subtraction (`-`) and division (`/`) operations. Python is also natively capable (i.e., without add-on libraries) of other mathematical operations, including those summarized in Table 2.

To expand on the above example we will now use the math module, which is provided by default in Python. A *module* is a self-contained collection of Python code that can be imported, via the `import` command, into any other Python program in order to provide some functionality to the runtime environment. (For instance, modules exist to parse protein sequence files, read PDB files or simulation trajectories, compute geometric properties, and so on. Much of Python's *extensibility* stems from the ability to use [and write] various modules, as presented in Supplemental Chapter 4 [ch04modules.py].) A collection of useful modules known as the *standard library* is bundled with Python, and can be relied upon as always being available to a Python program. Python's math module (in the standard library) introduces several mathematical capabilities, including one that is used in this section: `sin()`, which takes an angle in radians and outputs the sine of that angle. For example,

Table 2. Common mathematical operators in Python.

Symbol	Functionality
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo (yields remainder after division)
`//`	integer division (truncates toward zero)
`**`	exponentiation
`abs(a)`	absolute value of the number *a*, $\lvert a \rvert$
`math.sin(x)`	sine of *x* radians (other trigonometric functions are also available)
`math.factorial(n)`	factorial of *n*, *n*!
`math.log(a,b)`	$log_b(a)$ (defaults to natural logarithm, if no base *b* specified)
`math.sqrt(x)`	square root of *x*, $\sqrt{x}$

Common mathematical operators that are provided as built-in Python functions. Note that the behavior of `/` differs in versions of Python prior to 3.0; it previously acted as `//` does in recent versions of Python.

doi:10.1371/journal.pcbi.1004867.t002

```
1 import math
2 x = 21
3 y = math.sin(x)
4 print(y)
  0.8366556385360561
```

In the above program, the sine of `21 rad` is calculated, stored in `y`, and printed to the screen as the code's sole output. As in mathematics, an *expression* is formally defined as a unit of code that yields a value upon evaluation. As such, `x + 2*y`, `5 + 3`, `sin(pi)`, and even the number `5` alone, are examples of expressions (the final example is also known as a *literal*). All variable definitions involve setting a variable name equal to an expression.

Python's operator precedence rules mirror those in mathematics. For instance, `2+5*3` is interpreted as `2+(5*3)`. Python supports some operations that are not often found in arithmetic, such as `|` and `is`; a complete listing can be found in the official documentation [60]. Even complex expressions, like `x+3>>1|y&4>=5 or 6 == z+ x)`, are fully (unambiguously) resolved by Python's operator precedence rules. However, few programmers would have the patience to determine the meaning of such an expression by simple inspection. Instead, when expressions become complex, it is almost always a good idea to use parentheses to explicitly clarify the order: `(((x+3 >> 1) | y&4) >= 5) or (6 == (z + x))`.

The following block reveals an interesting deviation from the behavior of a variable as typically encountered in mathematics:

```
1 x = 5
2 x = 2
3 print(x)
  2
```

Viewed algebraically, the first two statements define an inconsistent system of equations (one with no solution) and may seem nonsensical. However, in Python, lines 1–2 are a perfectly valid pair of statements. When run, the `print` statement will display `2` on the screen. This occurs because Python, like most other languages, takes the statement `x = 2` to be a command to assign the value of `2` to `x`, ignoring any previous state of the variable `x`; such variable assignment statements are often denoted with the typographic convention "$x \leftarrow 2$". Lines 1–2 above are **instructions** to the Python interpreter, rather than some system of equations with no solutions for the variable *x*. This example also touches upon the fact that a Python variable is purely a **reference** to an object such as the integer `5` (For now, take an *object* to simply be an addressable chunk of memory, meaning it can have a value and be referenced by a variable; objects are further described in the section on OOP.). This is a property of Python's *type system*. Python is said to be *dynamically typed*, versus *statically typed* languages such as C. In statically typed languages, a program's data (variable names) are bound to both an object and a type, and type checking is performed at compile-time; in contrast, variable names in a program written in a dynamically typed language are bound only to objects, and type checking is performed at run-time. An extensive treatment of this topic can be found in [61]. Dynamic typing is illustrated by the following example. (The pound sign, #, starts a *comment*; Python ignores anything after a # sign, so in-line comments offer a useful mechanism for explaining and documenting one's code.)

```
1 x = 1        # x points to an integer object
2 y = x        # y points to the same object as does x (here, the integer 1)
3 x = "monty"  # x points to a different object now (a string)...
4 print(x)
  monty
5 print(y)      # ... but that does not alter y's reference to its object
  1
```

The above behavior results from the fact that, in Python, the notion of *type* (defined below) is attached to an object, not to any one of the potentially multiple names (variables) that reference that object. The first two lines illustrate that two or more variables can reference the same object (known as a *shared reference*), which in this case is of type `int`. When `y = x` is executed, `y` points to the object `x` points to (the integer `1`). When `x` is changed, `y` still points to that original integer object. Note that Python strings and integers are *immutable*, meaning they cannot be changed in-place. However, some other object types, such as lists (described below), are mutable. These aspects of the language can become rather subtle, and the various features of the variable/object relationship—shared references, object mutability, etc.—can give rise to complicated scenarios. Supplemental Chapter 8 (S1 Text) explores the Python memory model in more detail.

Statements and Types

A *statement* is a command that instructs the Python interpreter to **do** something. All expressions are statements, but a statement need not be an expression. For instance, a statement that, upon execution, causes a program to stop running would never return a value, so it cannot be an expression. Most broadly, statements are instructions, while expressions are combinations of symbols (variables, literals, operators, etc.) that evaluate to a particular value. This particular value might be numerical (e.g., `5`), a string (e.g., `'foo'`), Boolean (`True`/`False`), or some other type. Further distinctions between expressions and statements can become esoteric, and are not pertinent to much of the practical programming done in the biosciences.

The *type* of an object determines how the interpreter will treat the object when it is used. Given the code `x = 5`, we can say that "`x` is a variable that refers to an object that is of type `int`". We may simplify this to say "`x` is an `int`"; while technically incorrect, that is a shorter and more natural phrase. When the Python interpreter encounters the expression `x + y`, if `x` and `y` are [variables that point to objects of type] `int`, then the interpreter would use the addition hardware on the computer to add them. If, on the other hand, `x` and `y` were of type `str`, then Python would join them together. If one is a `str` and one is an `int`, the Python interpreter would "raise an exception" and the program would crash. Thus far, each variable we have encountered has been an integer (`int`) type, a string (`str`), or, in the case of `sin()`'s output, a real number stored to high precision (a `float`, for floating-point number). Strings and their constituent characters are among the most useful of Python's built-in types. Strings are sequences of characters, such as any word in the English language. In Python, a character is simply a string of length one. Each character in a string has a corresponding index, starting from `0` and ranging to index `n-1` for a string of *n* characters. Fig 1 diagrams the composition and some of the functionality of a string, and the following code-block demonstrates how to define and manipulate strings and characters:

```
x = "red"
y = "green"
z = "blue"
print(x + y + z)
redgreenblue
a = x[1]
b = y[2]
c = z[3]
print(a + " " + b + " " + c)
e e e
```

Here, three variables are created by assignment to three corresponding strings. The first `print` may seem unusual: the Python interpreter is instructed to "add" three strings; the

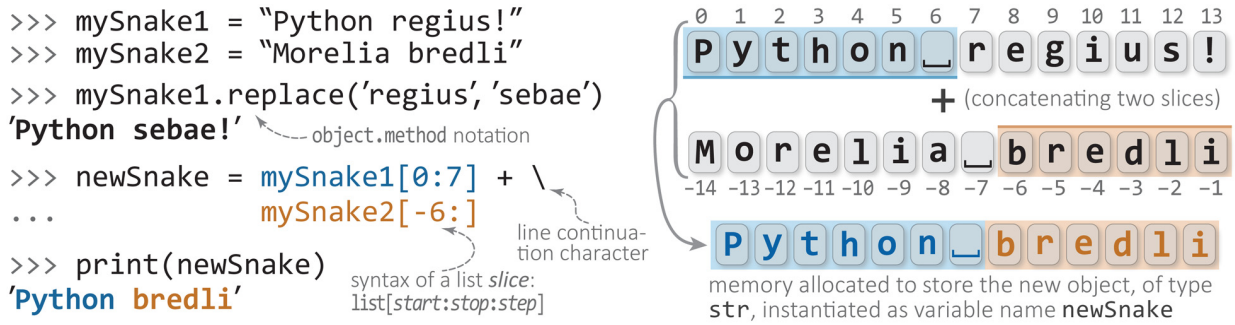

Fig 1. Strings in Python: anatomy and basic behavior. The anatomy and basic behavior of Python strings are shown, as samples of actual code (left panel) and corresponding conceptual diagrams (right panel). The Python interpreter prompts for user input on lines beginning with `>>>` (leftmost edge), while a starting . . . denotes a continuation of the previous line; output lines are not prefixed by an initial character (e.g., the fourth line in this example). Strings are simply character array objects (of type `str`), and a sample string-specific method (`replace`) is shown on line 3. As with ordinary lists, strings can be 'sliced' using the syntax shown here: the first list element to be included in the slice is indexed by `start`, and the last included element is at `stop-1`, with an optional stride of size `step` (defaults to one). Concatenation, via the `+` operator, is the joining of whole strings or subsets of strings that are generated via slicing (as in this case). For clarity, the integer indices of the string positions are shown only in the forward (left to right) direction for `mySnake1` and in the reverse direction for `mySnake2`. These two strings are sliced and concatenated to yield the object `newSnake`; note that slicing `mySnake1` as `[0:7]` and not `[0:6]` means that a whitespace char is included between the two words in the resultant `newSnake`, thus obviating the need for further manipulations to insert whitespace (e.g., concatenations of the form `word1+' '+word2`).

doi:10.1371/journal.pcbi.1004867.g001

interpreter joins them together in an operation known as *concatenation*. The second portion of code stores the character `'e'`, as extracted from each of the first three strings, in the respective variables, `a`, `b` and `c`. Then, their content is printed, just as the first three strings were. Note that spacing is not implicitly handled by Python (or most languages) so as to produce human-readable text; therefore, quoted whitespace was explicitly included between the strings (line 8; see also the underscore characters, '_', in Fig 1).

Exercise 1: Write a program to convert a temperature in degrees Fahrenheit to degrees Celsius and Kelvin. The topic of user input has not been covered yet (to be addressed in the section on File Management and I/O), so begin with a variable that you pre-set to the initial temperature (in °F). Your code should convert the temperature to these other units and `print` it to the console.

Functions

A deep benefit of the programming approach to problem-solving is that computers enable mechanization of repetitive tasks, such as those associated with data-analysis workflows. This is true in biological research and beyond. To achieve automation, a discrete and well-defined component of the problem-solving logic is encapsulated as a function. A *function* is a block of code that expresses the solution to a small, standalone problem/task; quite literally, a function can be any block of code that is defined by the user as being a function. Other parts of a program can then *call* the function to perform its task and possibly return a solution. For instance, a function can be repetitively applied to a series of input values via looping constructs (described below) as part of a data-processing pipeline.

Much of a program's versatility stems from its functions—the behavior and properties of each individual function, as well as the program's overall repertoire of available functions. Most simply, a function typically takes some values as its input *arguments* and acts on them; however, note that functions can be defined so as to not require any arguments (e.g., `print()` will give an empty line). Often, a function's arguments are specified simply by their position in the ordered list of arguments; e.g., the function is written such that the first expected argument

is height, the second is weight, etc. As an alternative to such a system of *positional arguments*, Python has a useful feature called *keyword arguments*, whereby one can name a function's arguments and provide them in any order, e.g. `plotData(dataset = dats, color = 'red', width = 10)`. Many scientific packages make extensive use of keyword arguments [62,63]. The arguments can be variables, explicitly specified values (constants, string literals, etc.), or even other functions. Most generally, **any** expression can serve as an argument (Supplemental Chapter 13 covers more advanced usage, such as function objects). Evaluating a function results in its *return value*. In this way, a function's arguments can be considered to be its domain and its return values to be its range, as for any mathematical function f that maps a domain X to the range Y, $X \xrightarrow{f} Y$. If a Python function is given arguments outside its domain, it may return an invalid/nonsensical result, or even crash the program being run. The following illustrates how to define and then call (*invoke*) a function:

```
1 def myFun(a,b):
2   c = a + b
3   d = a - b
4   return c*d   # NB: a return does not 'print' anything on its own
5 x = myFun(1,3) + myFun(2,8) + myFun(-1,18)
6 print(x)
  -391
```

To see the utility of functions, consider how much code would be required to calculate `x` (line 5) in the absence of any calls to `myFun`. Note that discrete chunks of code, such as the body of a function, are delimited in Python via whitespace, not curly braces, `{}`, as in C or Perl. In Python, each level of indentation of the source code corresponds to a separate *block* of statements that group together in terms of program logic. The first line of above code illustrates the syntax to declare a function: a function definition begins with the keyword `def`, the following word names the function, and then the names within parentheses (separated by commas) define the arguments to the function. Finally, a colon terminates the function definition. (Default values of arguments can be specified as part of the function definition; e.g., writing line 1 as `def myFun(a = 1,b = 3):` would set default values of `a` and `b`.) The three statements after `def myFun(a,b):` are indented by some number of spaces (two, in this example), and so these three lines (2–4) constitute a *block*. In this block, lines 2–3 perform arithmetic operations on the arguments, and the final line of this function specifies the return value as the product of variables `c` and `d`. In effect, a `return` statement is what the function evaluates to when called, this return value taking the place of the original function call. It is also possible that a function returns nothing at all; e.g., a function might be intended to perform various manipulations and not necessarily return any output for downstream processing. For example, the following code defines (and then calls) a function that simply prints the values of three variables, without a `return` statement:

```
1 def readOut(a,b,c):
2   print("Variable 1 is: ", a)
3   print("Variable 2 is: ", b)
4   print("Variable 3 is: ", c)
5 readOut(1,2,4)
  Variable 1 is: 1
  Variable 2 is: 2
  Variable 3 is: 4
6 readOut(21,5553,3.33)
  Variable 1 is : 21
  Variable 2 is : 5553
  Variable 3 is : 3.33
```

Code Organization and Scope

Beyond automation, structuring a program into functions also aids the modularity and interpretability of one's code, and ultimately facilitates the debugging process—an important consideration in all programming projects, large or small.

Python functions can be *nested*; that is, one function can be defined inside another. If a particular function is needed in only one place, it can be defined where it is needed and it will be unavailable elsewhere, where it would not be useful. Additionally, nested function definitions have access to the variables that are available when the nested function is defined. Supplemental Chapter 13 explores nested functions in greater detail. A function is an object in Python, just like a string or an integer. (Languages that allow function names to behave as objects are said to have "first-class functions.") Therefore, a function can itself serve as an argument to another function, analogous to the mathematical composition of two functions, $g(f(x))$. This property of the language enables many interesting programming techniques, as explored in Supplemental Chapters 9 and 13.

A variable created inside a block, e.g. within a function, cannot be accessed by name from outside that block. The variable's *scope* is limited to the block wherein it was defined. A variable or function that is defined outside of every other block is said to be *global* in scope. Variables can appear within the scope in which they are defined, or any block within that scope, but the reverse is not true: variables cannot escape their scope. This rule hierarchy is diagrammed in Fig 2. There is only one global scope, and variables in it necessarily "persist" between function calls (unlike variables in local scope). For instance, consider two functions, `fun1` and `fun2`; for convenience, denote their local scopes as ℓ_1 and ℓ_2, and denote the global scope as $\mathcal{G}$. Starting in $\mathcal{G}$, a call to `fun1` places us in scope ℓ_1. When `fun1` successfully returns, we return to scope $\mathcal{G}$; a call to `fun2` places us in scope ℓ_2, and after it completes we return yet again to $\mathcal{G}$. We

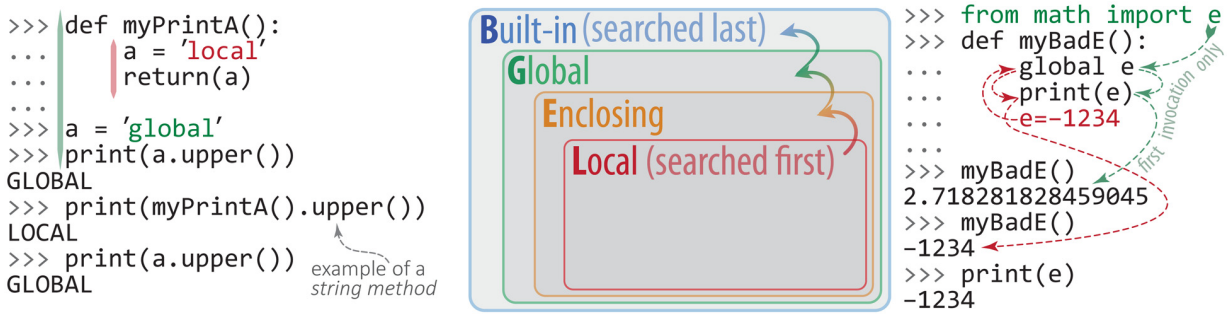

Fig 2. Python's scope hierarchy and variable name resolution. As described in the text, multiple names (variables) can reference a single object. Conversely, can a single variable, say `x`, reference multiple objects in a unique and well-defined manner? Exactly this is enabled by the concept of a *namespace*, which can be viewed as the set of all name↔object mappings for all variable names and objects at a particular "level" in a program. This is a crucial concept, as everything in Python is an object. The key idea is that name↔object mappings are insulated from one another, and therefore free to vary, at different "levels" in a program—e.g., `x` might refer to object `obj2` in a block of code buried (many indentation levels deep) within a program, whereas the same variable name `x` may reference an entirely different object, `obj1`, when it appears as a top-level (module-level) name definition. This seeming ambiguity is resolved by the notion of variable scope. The term *scope* refers to the level in the namespace hierarchy that is searched for name↔object mappings; different mappings can exist in different scopes, thus avoiding potential name collisions. At a specific point in a block of code, in what order does Python search the namespace levels? (And, which of the potentially multiple name↔object mappings takes precedence?) Python resolves variable names by traversing scope in the order L→E→G→B, as shown here. L stands for the **l**ocal, innermost scope, which contains local names and is searched first; E follows, and is the scope of any **e**nclosing functions; next is G, which is the namespace of all **g**lobal names in the currently loaded modules; finally, the outermost scope B, which consists of Python's **b**uilt-in names (e.g., `int`), is searched last. The two code examples in this figure demonstrate variable name resolution at local and global scope levels. In the code on the right-hand side, the variable `e` is used both (i) as a name imported from the math module (global scope) and (ii) as a name that is local to a function body, albeit with the `global` keyword prior to being assigned to the integer `-1234`. This construct leads to a confusing flow of logic (colored arrows), and is considered poor programming practice.

doi:10.1371/journal.pcbi.1004867.g002

always return to $\mathcal{G}$. In this sense, local scope varies, whereas global scope (by definition) persists between function calls, is available inside/outside of functions, etc. Explicitly tracking the precise scope of every object in a large body of code can be cumbersome. However, this is rarely burdensome in practice: Variables are generally defined (and are therefore in scope) where they are used. After encountering some out-of-scope errors and gaining experience with nested functions and variables, carefully managing scope in a consistent and efficient manner will become an implicit skill (and will be reflected in one's coding style).

Well-established practices have evolved for structuring code in a logically organized (often hierarchical) and "clean" (lucid) manner, and comprehensive treatments of both practical and abstract topics are available in numerous texts. See, for instance, the practical guide *Code Complete*[64], the intermediate-level *Design Patterns: Elements of Reusable Object-Oriented Software*[65], and the classic (and more abstract) texts *Structure and Interpretation of Computer Programs*[39] and *Algorithms*[50]; a recent, and free, text in the latter class is *Introduction to Computing*[40]. Another important aspect of coding is closely related to the above: usage of brief, yet informative, names as identifiers for variables and function definitions. Even a mid-sized programming project can quickly grow to thousands of lines of code, employ hundreds of functions, and involve hundreds of variables. Though the fact that many variables will lie outside the scope of one another lessens the likelihood of undesirable references to ambiguous variable names, one should note that careless, inconsistent, or undisciplined nomenclature **will** confuse later efforts to understand a piece of code, for instance by a collaborator or, after some time, even the original programmer. Writing clear, well-defined and well-annotated code is an essential skill to develop. Table 3 outlines some suggested naming practices.

Python minimizes the problems of conflicting names via the concept of namespaces. A *namespace* is the set of all possible (valid) names that can be used to uniquely identify an object at a given level of scope, and in this way it is a more generalized concept than scope (see also Fig 2).

Table 3. Sample variable-naming schemes in Python.

Variable Name	Valid?	Comments
`numHelix`	yes	By using the camelCase convention (capitalize the first letter of every word following the first), this variable name is reasonably descriptive, easily read, and brief.
`123myVar!?`	no	This variable uses forbidden characters ('!' and '?'), and it also starts with digits; all special characters apart from '_' are disallowed, and no variable can begin with a digit.
`int`	yes	This Python built-in function is technically allowed as a variable name; however, its attempted usage may yield unexpected behavior, as the statement '`int(x)`' converts an object '`x`' to type `int`.
`else`	no	This is an element of the basic syntax in Python (a *keyword*). Using a keyword as a variable name is a syntax error. Other common keywords are `for`, `if`, `not`, `return`, and `def`.
`_myVar2`	yes	Though fairly nondescript, this variable does not contain forbidden characters and is technically valid (note that the underscore character is generally used to denote [protected] member variables in objects).
`IlI1lI`	yes	While technically valid, such a name is unnecessarily frustrating for those reading and maintaining code.

These examples of variable-naming schemes in Python are annotated with comments as to their validity and their suitability (the latter is more subjective). In practice, variables at higher (broader) levels of scope tend to have names that are longer, more descriptive, and less ambiguous than at lower (narrower) levels of scope.

doi:10.1371/journal.pcbi.1004867.t003

To access a name in a different namespace, the programmer must tell the interpreter what namespace to search for the name. An `import`ed module, for example, creates its own new namespace. The math module creates a namespace (called `math`) that contains the `sin()` function. To access `sin()`, the programmer must qualify the function call with the namespace to search, as in `y = math.sin(x)`. This precision is necessary because merging two namespaces that might possibly contain the same names (in this case, the `math` namespace and the global namespace) results in a *name collision*. Another example would be to consider the files in a Unix directory (or a Windows folder); in the namespace of this top-level directory, one file can be named foo1 and another foo2, but there cannot be two files named foo—that would be a name collision.

Exercise 2: Recall the temperature conversion program of Exercise 1. Now, write a function to perform the temperature conversion; this function should take one argument (the input temperature). To test your code, use the function to convert and print the output for some arbitrary temperatures of your choosing.

Control Flow: Conditionals

> "Begin at the beginning," the King said gravely, "and go on till you come to the end; then, stop."
>
> —Lewis Carroll, *Alice in Wonderland*

Thus far, all of our sample code and exercises have featured a linear flow, with statements executed and values emitted in a predictable, deterministic manner. However, most scientific datasets are not amenable to analysis via a simple, predefined stream of instructions. For example, the initial data-processing stages in many types of experimental pipelines may entail the assignment of statistical confidence/reliability scores to the data, and then some form of decision-making logic might be applied to filter the data. Often, **if** a particular datum does not meet some statistical criterion and is considered a likely outlier, **then** a special task is performed; **otherwise**, another (default) route is taken. This branched **if**–**then**–**else** logic is a key decision-making component of virtually any algorithm, and it exemplifies the concept of control flow. The term *control flow* refers to the progression of logic as the Python interpreter traverses the code and the program "runs"—transitioning, as it runs, from one state to the next, choosing which statements are executed, iterating over a loop some number of times, and so on. (Loosely, the *state* can be taken as the line of code that is being executed, along with the collection of all variables, and their values, accessible to a running program at any instant; given the precise state, the next state of a deterministic program can be predicted with perfect precision.) The following code introduces the `if` statement:

```
1 from random import randint
2 a = randint(0,100)   # get a random integer between 0 and 100 (inclusive)
3 if(a < 50):
4   print("variable is less than 50")
5 else:
6   print("the variable is not less than 50")
  variable is less than 50
```

In this example, a random integer between `0` and `100` is assigned to the variable `a`. (Though not applicable to `randint`, note that many sequence/list-related functions, such as `range(a,b)`, generate collections that start at the first argument and end just before the last argument. This is because the function `range(a,b)` produces $b - a$ items starting at a; with a default stepsize of one, this makes the endpoint `b-1`.) Next, the `if` statement tests whether

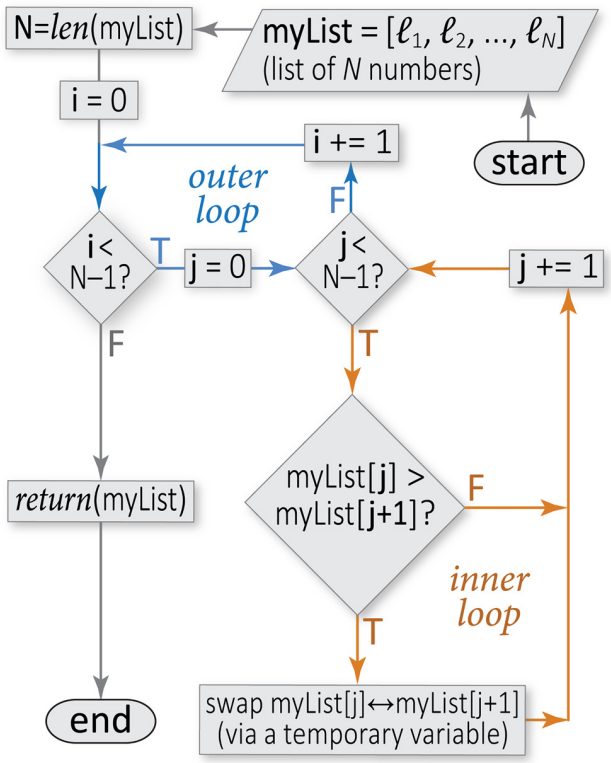

Fig 3. Sample flowchart for a sorting algorithm. This flowchart illustrates the conditional constructs, loops, and other elements of control flow that comprise an algorithm for sorting, from smallest to largest, an arbitrary list of numbers (the algorithm is known as "bubble sort"). In this type of diagram, arrows symbolize the flow of logic (control flow), rounded rectangles mark the start and end points, slanted parallelograms indicate I/O (e.g., a user-provided list), rectangles indicate specific subroutines or procedures (blocks of statements), and diamonds denote conditional constructs (branch points). Note that this sorting algorithm involves a pair of nested loops over the list size (blue and orange), meaning that the calculation cost will go as the square of the input size (here, an *N*-element list); this cost can be halved by adjusting the inner loop conditional to be "$j < N - i - 1$", as the largest *i* elements will have already reached their final positions.

doi:10.1371/journal.pcbi.1004867.g003

the variable is less than `50`. If that condition is unfulfilled, the block following `else` is executed. Syntactically, `if` is immediately followed by a *test condition*, and then a colon to denote the start of the `if` statement's block (Fig 3 illustrates the use of conditionals). Just as with functions, the further indentation on line 4 creates a block of statements that are executed together (here, the block has only one statement). Note that an `if` statement can be defined without a corresponding `else` block; in that case, Python simply continues executing the code that is indented by one less level (i.e., at the same indentation level as the `if` line). Also, Python offers a built-in `elif` keyword (a contraction of "else if") that tests a subsequent conditional if and only if the first condition is not met. A series of `elif` statements can be used to achieve similar effects as the `switch/case` statement constructs found in C and in other languages (including Unix shell scripts) that are often encountered in bioinformatics.

Now, consider the following extension to the preceding block of code. Is there any fundamental issue with it?

```
1 from random import randint
2 a = randint(0,100)
3 if(a < 50):
```

```
4   print("variable is less than 50")
5 if(a > 50):
6   print("variable is greater than 50")
7 else:
8   print("the variable must be 50")
  variable is greater than 50
```

This code will function as expected for `a = 50`, as well as values exceeding `50`. However, for `a` less than `50`, the print statements will be executed from **both** the less-than (line 4) and equal-to (line 8) comparisons. This erroneous behavior results because an `else` statement is bound solely to the `if` statement that it directly follows; in the above code-block, an `elif` would have been the appropriate keyword for line 5. This example also underscores the danger of assuming that lack of a certain condition (a `False` built-in Boolean type) necessarily implies the fulfillment of a second condition (a `True`) for comparisons that seem, at least superficially, to be linked. In writing code with complicated streams of logic (conditionals and beyond), robust and somewhat redundant logical tests can be used to mitigate errors and unwanted behavior. A strategy for building streams of conditional statements into code, and for debugging existing codebases, involves (i) outlining the range of possible inputs (and their expected outputs), (ii) crafting the code itself, and then (iii) testing each possible type of input, carefully tracing the logical flow executed by the algorithm against what was originally anticipated. In step (iii), a careful examination of "edge cases" can help debug code and pinpoint errors or unexpected behavior. (In software engineering parlance, *edge cases* refer to extreme values of parameters, such as minima/maxima when considering ranges of numerical types. Recognition of edge-case behavior is useful, as a disproportionate share of errors occur near these cases; for instance, division by zero can crash a function if the denominator in each division operation that appears in the function is not carefully checked and handled appropriately. Though beyond the scope of this primer, note that Python supplies powerful error-reporting and exception-handling capabilities; see, for instance, *Python Programming*[66] for more information.) Supplemental Chapters 14 and 16 in S1 Text provide detailed examples of testing the behavior of code.

Exercise 3: Recall the temperature-conversion program designed in Exercises 1 and 2. Now, rewrite this code such that it accepts two arguments: the initial temperature, and a letter designating the units of that temperature. Have the function convert the input temperature to the alternative scale. If the second argument is `'C'`, convert the temperature to Fahrenheit, if that argument is `'F'`, convert it to Celsius.

Integrating what has been described thus far, the following example demonstrates the power of control flow—not just to define computations in a structured/ordered manner, but also to solve real problems by devising an algorithm. In this example, we sort three randomly chosen integers:

```
1 from random import randint
2 def numberSort():
3   a = randint(0,100)
4   b = randint(0,100)
5   c = randint(0,100)
6   # reminder: text following the pound sign is a comment in Python.
7   # begin sort; note the nested conditionals here
8   if ((a > b) and (a > c)):
9     largest = a
10     if(b > c):
11       second = b
12       third = c
13     else:
```

```
14      second = c
15      third = b
16   # a must not be largest
17   elif(b > c):
18     largest = b
19     if(c > a):
20       second = c
21       third = a
22     else:
23       second = a
24       third = c
25   # a and b are not largest, thus c must be
26   else:
27     largest = c
28     if(b < a):
29       second = a
30       third = b
31     else:
32       second = b
33       third = a
34   # Python's assert function can be used for sanity checks.
35   # If the argument to assert() is False, the program will crash.
36   assert(largest > second)
37   assert(second > third)
38   print("Sorted:", largest, ",", second, ",", third)
39 numberSort()
  Sorted: 50, 47, 11
```

Control Flow: Repetition via While Loops

Whereas the `if` statement tests a condition exactly once and branches the code execution accordingly, the `while` statement instructs an enclosed block of code to repeat so long as the given condition (the *continuation condition*) is satisfied. In fact, `while` can be considered as a repeated `if`. This is the simplest form of a loop, and is termed a *while loop* (Fig 3). The condition check occurs once before entering the associated block; thus, Python's `while` is a *pre-test* loop. (Some languages feature looping constructs wherein the condition check is performed after a first iteration; C's *do–while* is an example of such a *post-test* loop. This is mentioned because looping constructs should be carefully examined when comparing source code in different languages.) If the condition is true, the block is executed and then the interpreter effectively jumps to the `while` statement that began the block. If the condition is false, the block is skipped and the interpreter jumps to the first statement after the block. The code below is a simple example of a while loop, used to generate a counter that prints each integer between `1` and `100` (inclusive):

```
1 counter = 1
2 while(counter <= 100):
3   print(counter)
4   counter = counter + 1
  1
  2
  ...
  99
  100
5 print("done!")
  done!
```

This code will begin with a variable, then print and increment it until its value is `101`, at which point the enclosing while loop ends and a final string (line 5) is printed. Crucially, one should verify that the loop termination condition can, in fact, be reached. If not—e.g., if the loop were specified as `while(True):` for some reason—then the loop would continue indefinitely, creating an infinite loop that would render the program unresponsive. (In many environments, such as a Unix shell, the keystroke `Ctrl-c` can be used as a *keyboard interrupt* to break out of the loop.)

Exercise 4: With the above example as a starting point, write a function that chooses two randomly-generated integers between `0` and `100`, inclusive, and then prints all numbers between these two values, counting from the lower number to the upper number.

Recursion

"In order to understand recursion, you must first understand recursion."
— Anonymous

Recursion is a subtle concept. A while loop is conceptually straightforward: a block of statements comprising the body of the loop is repeatedly executed as long as a condition is true. A *recursive* function, on the other hand, calls **itself** repeatedly, effectively creating a loop. Recursion is the most natural programming approach (or *paradigm*) for solving a complex problem that can be decomposed into (easier) subproblems, each of which resembles the overall problem. Mathematically, problems that are formulated in this manner are known as *recurrence relations*, and a classic example is the factorial (below). Recursion is so fundamental and general a concept that iterative constructs (`for`, `while` loops) can be expressed recursively; in fact, some languages dispense with loops entirely and rely on recursion for all repetition. The key idea is that a recursive function calls itself from within its own function body, thus progressing one step closer to the final solution at each self-call. The recursion terminates once it has reached a trivially simple final operation, termed the *base case*. (Here, the word "simple" means only that evaluation of the final operation yields no further recursive steps, with no implication as to the computational complexity of that final operation.) Calculation of the factorial function, $f(n) = n!$, is a classic example of a problem that is elegantly coded in a recursive manner. Recall that the factorial of a natural number, n, is defined as:

$$n! = \begin{cases} 1 & n = 1 \quad \text{(base case)} \\ n * (n-1)! & n > 1 \end{cases} \tag{1}$$

This function can be compactly implemented in Python like so:

```
1 def factorial(n):
2   assert(n > 0)   # Crash on invalid input
3   if(n == 1):
4     return 1
5   else:
6     return n * factorial(n-1)
```

A call to this `factorial` function will return `1` if the input is equal to one, and otherwise will return the input value multiplied by the factorial of that integer less one (`factorial(n-1)`). Note that this recursive implementation of the factorial perfectly matches its mathematical definition. This often holds true, and many mathematical operations on data are most easily expressed recursively. When the Python interpreter encounters the call to the `factorial` function within the function block itself (line 6), it generates a

new instance of the function on the fly, while retaining the original function in memory (technically, these function instances occupy the runtime's *call stack*). Python places the current function call on hold in the call stack while the newly-called function is evaluated. This process continues until the base case is reached, at which point the function returns a value. Next, the previous function instance in the call stack resumes execution, calculates its result, and returns it. This process of traversing the call stack continues until the very first invocation has returned. At that point, the call stack is empty and the function evaluation has completed.

Expressing Problems Recursively

Defining recursion simply as a function calling itself misses some nuances of the recursive approach to problem-solving. Any difficult problem (e.g., $f(n) = n!$) that can be expressed as a simpler instance of the same problem (e.g., $f(n) = n^{*}f(n-1)$) is amenable to a recursive solution. Only when the problem is trivially easy (1!, `factorial(1)` above) does the recursive solution give a direct (one-step) answer. Recursive approaches fundamentally differ from more iterative (also known as *procedural*) strategies: Iterative constructs (loops) express the entire solution to a problem in more explicit form, whereas recursion repeatedly makes a problem simpler until it is trivial. Many data-processing functions are most naturally and compactly solved via recursion.

The recursive descent/ascent behavior described above is extremely powerful, and care is required to avoid pitfalls and frustration. For example, consider the following addition algorithm, which uses the equality operator (==) to test for the base case:

```
1 def badRecursiveAdder(x):
2   if(x == 1):
3     return x
4   else:
5     return x + badRecursiveAdder(x-2)
```

This function does include a base case (lines 2–3), and at first glance may seem to act as expected, yielding a sequence of squares (1, 4, 9, 16. . .) for $x = 1, 3, 5, 7, \ldots$ Indeed, for odd `x` greater than `1`, the function will behave as anticipated. However, if the argument is negative or is an even number, the base case will never be reached (note that line 5 subtracts `2`), causing the function call to simply hang, as would an infinite loop. (In this scenario, Python's *maximum recursion depth* will be reached and the call stack will overflow.) Thus, in addition to defining the function's base case, it is also crucial to confirm that **all** possible inputs will reach the base case. A valid recursive function must progress towards—and eventually reach—the base case with every call. More information on recursion can be found in Supplemental Chapter 7 in S1 Text, in Chapter 4 of [40], and in most computer science texts.

Exercise 5: Consider the Fibonacci sequence of integers, 0, 1, 1, 2, 3, 5, 8, 13, . . ., given by

$$F_n = \begin{cases} n & n \leq 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases} \tag{2}$$

This sequence appears in the study of phyllotaxis and other areas of biological pattern formation (see, e.g., [67]). Now, write a recursive Python function to compute the nth Fibonacci number, F_n, and test that your program works. Include an `assert` to make sure the argument is positive. Can you generalize your code to allow for different seed values ($F_0 = l$, $F_1 = m$, for integers l and m) as arguments to your function, thereby creating new sequences? (Doing so gets you one step closer to Lucas sequences, L_n, which are a highly general class of recurrence relations.)

Exercise 6: Many functions can be coded both recursively and iteratively (using loops), though often it will be clear that one approach is better suited to the given problem (the factorial is one such example). In this exercise, devise an **iterative** Python function to compute the factorial of a user-specified integer argument. As a bonus exercise, try coding the Fibonacci sequence in iterative form. Is this as straightforward as the recursive approach? Note that Supplemental Chapter 7 in the S1 Text might be useful here.

Data Collections: Tuples, Lists, For Loops, and Dictionaries

A staggering degree of algorithmic complexity is possible using only variables, functions, and control flow concepts. However, thus far, numbers and strings are the only data types that have been discussed. Such data types can be used to represent protein sequences (a string) and molecular masses (a floating point number), but actual scientific data are seldom so simple! The data from a mass spectrometry experiment are a list of intensities at various m/z values (the mass spectrum). Optical microscopy experiments yield thousands of images, each consisting of a large two-dimensional array of pixels, and each pixel has color information that one may wish to access [68]. A protein multiple sequence alignment can be considered as a two-dimensional array of characters drawn from a 21-letter alphabet (one letter per amino acid (AA) and a gap symbol), and a protein 3D structural alignment is even more complex. Phylogenetic trees consist of sets of species, individual proteins, or other taxonomic entities, organized as (typically) binary trees with branch weights that represent some metric of evolutionary distance. A trajectory from an MD or Brownian dynamics simulation is especially dense: Cartesian coordinates and velocities are specified for upwards of 10^6 atoms at $>10^6$ time-points (every ps in a μs-scale trajectory). As illustrated by these examples, real scientific data exhibit a level of complexity far beyond Python's relatively simple built-in data types. Modern datasets are often quite heterogeneous, particularly in the biosciences [69], and therefore *data abstraction* and *integration* are often the major goals. The data challenges hold true at all levels, from individual RNA transcripts [70] to whole bacterial cells [71] to biomedical informatics [72].

In each of the above examples, the relevant data comprise a collection of entities, each of which, in turn, is of some simpler data type. This unifying principle offers a way forward. The term *data structure* refers to an object that stores data in a specifically organized (structured) manner, as defined by the programmer. Given an adequately well-specified/defined data structure, arbitrarily complex collections of data can be readily handled by Python, from a simple array of integers to a highly intricate, multi-dimensional, heterogeneous (mixed-type) data structure. Python offers several built-in sequence data structures, including strings, lists, and tuples.

Tuples

A *tuple* (pronounced like "couple") is simply an ordered sequence of objects, with essentially no restrictions as to the types of the objects. Thus, the tuple is especially useful in building data structures as higher-order collections. Data that are inherently sequential (e.g., time-series data recorded by an instrument) are naturally expressed as a tuple, as illustrated by the following syntactic form: `myTuple = (0,1,3)`. The tuple is surrounded by parentheses, and commas separate the individual elements. The empty tuple is denoted `()`, and a tuple of one element contains a comma after that element, e.g., `(1,)`; the final comma lets Python distinguish between a tuple and a mathematical operation. That is, `2*(3+1)` must not treat `(3+1)` as a tuple. A parenthesized expression is therefore not made into a tuple unless it contains commas. (The `type` function is a useful built-in function to probe an object's type. At

the Python interpreter, try the statements `type((1))` and `type((1,))`. How do the results differ?)

A tuple can contain any sort of object, including another tuple. For example, `diverseTuple = (15.38,"someString",(0,1))` contains a floating-point number, a string, and another tuple. This versatility makes tuples an effective means of representing complex or heterogeneous data structures. Note that any component of a tuple can be referenced using the same notation used to index individual characters within a string; e.g., `diverseTuple[0]` gives `15.38`.

In general, data are optimally stored, analyzed, modified, and otherwise processed using data structures that reflect any underlying structure of the data itself. Thus, for example, two-dimensional datasets are most naturally stored as tuples of tuples. This abstraction can be taken to arbitrary depth, making tuples useful for storing arbitrarily complex data. For instance, tuples have been used to create generic tensor-like objects. These rich data structures have been used in developing new tools for the analysis of MD trajectories [18] and to represent biological sequence information as hierarchical, multidimensional entities that are amenable to further processing in Python [20].

As a concrete example, consider the problem of representing signal intensity data collected over time. If the data are sampled with perfect periodicity, say every second, then the information could be stored (most compactly) in a one-dimensional tuple, as a simple succession of intensities; the index of an element in the tuple maps to a time-point (index `0` corresponds to the measurement at time t_0, index `1` is at time t_1, etc.). What if the data were sampled unevenly in time? Then each datum could be represented as an ordered pair, $(t, I(t))$, of the intensity I at each time-point t; the full time-series of measurements is then given by the sequence of 2-element tuples, like so:

```
dataSet = ((0.00,0.2),
           (0.17,0.3),
           (0.33,0.4),
           (0.40,0.2),
           (0.90,0.0))

print(dataSet[4])       # prints the final ordered pair
(0.9, 0.0)
print(dataSet[0][1])   # prints the intensity at timestep zero
0.2
print(dataSet[2][0])   # prints the time at timestep two
0.33
```

Three notes concern the above code: (i) From this two-dimensional data structure, the syntax `dataSet[i][j]` retrieves the jth element from the ith tuple. (ii) Negative indices can be used as shorthand to index from the end of most collections (tuples, lists, etc.), as shown in Fig 1; thus, in the above example `dataSet[-1]` represents the same value as `dataSet[4]`. (iii) Recall that Python treats all lines of code that belong to the same block (or degree of indentation) as a single unit. In the example above, the first line alone is not a valid (closed) expression, and Python allows the expression to continue on to the next line; the lengthy `dataSet` expression was formatted as above in order to aid readability.

Once defined, a tuple cannot be altered; tuples are said to be *immutable* data structures. This rigidity can be helpful or restrictive, depending on the context and intended purpose. For instance, tuples are suitable for storing numerical constants, or for ordered collections that are generated once during execution and intended only for referencing thereafter (e.g., an input stream of raw data).

Lists

A mutable data structure is the Python *list*. This built-in sequence type allows for the addition, removal, and modification of elements. The syntactic form used to define lists resembles the definition of a tuple, except that the parentheses are replaced with square brackets, e.g. `myList = [0, 1, 42, 78]`. (A trailing comma is unnecessary in one-element lists, as `[1]` is unambiguously a list.) As suggested by the preceding line, the elements in a Python list are typically more homogeneous than might be found in a tuple: The statement `myList2 = ['a',1]`, which defines a list containing both string and numeric types, is technically valid, but `myList2 = ['a','b']` or `myList2 = [0, 1]` would be more frequently encountered in practice. Note that `myList[1] = 3.14` is a perfectly valid statement that can be applied to the already-defined object named `myList` (as long as `myList` already contains two or more elements), resulting in the modification of the second element in the list. Finally, note that `myList[5] = 3.14` will raise an error, as the list defined above does not contain a sixth element. The index is said to be *out of range*, and a valid approach would be to append the value via `myList.append(3.14)`.

The foregoing description only scratches the surface of Python's built-in data structures. Several functions and methods are available for lists, tuples, strings, and other built-in types. For lists, `append`, `insert`, and `remove` are examples of oft-used methods; the function `len()` returns the number of items in a sequence or collection, such as the length of a string or number of elements in a list. All of these "list methods" behave similarly as any other function—arguments are generally provided as input, some processing occurs, and values may be returned. (The OOP section, below, elaborates the relationship between functions and methods.)

Iteration with For Loops

Lists and tuples are examples of *iterable* types in Python, and the *for loop* is a useful construct in handling such objects. (Custom iterable types are introduced in Supplemental Chapter 17 in S1 Text.) A Python `for` loop iterates over a collection, which is a common operation in virtually all data-analysis workflows. Recall that a `while` loop requires a counter to track progress through the iteration, and this counter is tested against the continuation condition. In contrast, a `for` loop handles the count implicitly, given an argument that is an iterable object:

```
1 myData = [1.414, 2.718, 3.142, 4.669]
2 total = 0
3 for datum in myData:
4   # the next statement uses a compound assignment operator; in
5   # the addition assignment operator, a += b means a = a + b
6   total += datum
7   print("added " + str(datum) + " to sum.")
8   # str makes a string from datum so we can concatenate with +.
  added 1.414 to sum.
  added 2.718 to sum.
  added 3.142 to sum.
  added 4.669 to sum.
9 print(total)
  11.942999999999998
```

In the above loop, all elements in `myData` are of the same type (namely, floating-point numbers). This is not mandatory. For instance, the heterogeneous object `myData = ['a','b',1,2]` is iterable, and therefore it is a valid argument to a `for` loop (though not the above loop, as string and integer types cannot be mixed as operands to the + operator). The context dependence of the + symbol, meaning either numeric addition or a concatenation

operator, depending on the arguments, is an example of *operator overloading*. (Together with dynamic typing, operator overloading helps make Python a highly expressive programming language.) In each iteration of the above loop, the variable `datum` is assigned each successive element in `myData`; specifying this iterative task as a `while` loop is possible, but less straightforward. Finally, note the syntactic difference between Python's `for` loops and the `for (<initialize>; <condition>; <update>) {<body>}` construct that is found in C, Perl, and other languages encountered in computational biology.

Exercise 7: Consider the fermentation of glucose into ethanol: $C_6H_{12}O_6 \rightarrow 2C_2H_5OH + 2CO_2$. A fermentor is initially charged with 10,000 liters of feed solution and the rate of carbon dioxide production is measured by a sensor in moles/hour. At t = 10, 20, 30, 40, 50, 60, 70, and 80 hours, the CO_2 generation rates are 58.2, 65.2, 67.8, 65.4, 58.8, 49.6, 39.1, and 15.8 moles/hour respectively. Assuming that each reading represents the average CO_2 production rate over the previous ten hours, calculate the total amount of CO_2 generated and the final ethanol concentration in grams per liter. Note that Supplemental Chapters 6 and 9 might be useful here.

Exercise 8: Write a program to compute the distance, $d(\mathbf{r_1}, \mathbf{r_2})$, between two arbitrary (user-specified) points, $\mathbf{r_1} = (x_1, y_1, z_1)$ and $\mathbf{r_2} = (x_2, y_2, z_2)$, in 3D space. Use the usual Euclidean distance between two points—the straight-line, "as the bird flies" distance. Other distance metrics, such as the Mahalanobis and Manhattan distances, often appear in computational biology too. With your code in hand, note the ease with which you can adjust your entire data-analysis workflow simply by modifying a few lines of code that correspond to the definition of the distance function. As a bonus exercise, generalize your code to read in a list of points and compute the total path length. Supplemental Chapters 6, 7, and 9 might be useful here.

Sets and Dictionaries

Whereas lists, tuples, and strings are ordered (sequential) data types, Python's *sets* and *dictionaries* are unordered data containers. Dictionaries, also known as *associative arrays* or *hashes* in Perl and other common languages, consist of key:value pairs enclosed in braces. They are particularly useful data structures because, unlike lists and tuples, the values are not restricted to being indexed solely by the integers corresponding to sequential position in the data series. Rather, the keys in a dictionary serve as the index, and they can be of any immutable data type (strings, numbers, or tuples of immutable data). A simple example, indexing on three-letter abbreviations for amino acids and including molar masses, would be `aminoAcids = {'ala':('a','alanine', 89.1),'cys':('c','cysteine', 121.2)}`. A dictionary's items are accessed via square brackets, analogously as for a tuple or list, e.g., `aminoAcids['ala']` would retrieve the tuple `('a','alanine', 89.1)`. As another example, dictionaries can be used to create lookup tables for the properties of a collection of closely related proteins. Each key could be set to a unique identifier for each protein, such as its UniProt ID (e.g., Q8ZYG5), and the corresponding values could be an intricate tuple data structure that contains the protein's isoelectric point, molecular weight, PDB accession code (if a structure exists), and so on. Dictionaries are described in greater detail in Supplemental Chapter 10 in the S1 Text.

Further Data Structures: Trees and Beyond

Python's built-in data structures are made for sequential data, and using them for other purposes can quickly become awkward. Consider the task of representing genealogy: an individual may have some number of children, and each child may have their own children, and so on. There is no straightforward way to represent this type of information as a list or tuple. A better

approach would be to represent each organism as a tuple containing its children. Each of those elements would, in turn, be another tuple with children, and so on. A specific organism would be a *node* in this data structure, with a *branch* leading to each of its *child* nodes; an organism having no children is effectively a *leaf*. A node that is not the child of any other node would be the *root* of this tree. This intuitive description corresponds, in fact, to exactly the terminology used by computer scientists in describing *trees* [73]. Trees are pervasive in computer science. This document, for example, could be represented purely as a list of characters, but doing so neglects its underlying structure, which is that of a tree (sections, sub-sections, sub-sub-sections, . . .). The whole document is the root entity, each section is a node on a branch, each subsection a branch from a section, and so on down through the paragraphs, sentences, words, and letters. A common and intuitive use of trees in bioinformatics is to represent phylogenetic relationships. However, trees are such a general data structure that they also find use, for instance, in computational geometry applications to biomolecules (e.g., to optimally partition data along different spatial dimensions [74,75]).

Trees are, by definition, (i) *acyclic*, meaning that following a branch from node *i* will never lead back to node *i*, and any node has exactly one parent; and *(ii) directed*, meaning that a node knows only about the nodes "below" it, not the ones "above" it. Relaxing these requirements gives a *graph* [76], which is an even more fundamental and universal data structure: A graph is a set of vertices that are connected by edges. Graphs can be subtle to work with and a number of clever algorithms are available to analyze them [77].

There are countless data structures available, and more are constantly being devised. Advanced examples range from the biologically-inspired neural network, which is essentially a graph wherein the vertices are linked into communication networks to emulate the neuronal layers in a brain [78], to very compact probabilistic data structures such as the Bloom filter [79], to self-balancing trees [80] that provide extremely fast insertion and removal of elements for performance-critical code, to copy-on-write B-trees that organize terabytes of information on hard drives [81].

Object-Oriented Programming in a Nutshell: Classes, Objects, Methods, and All That

OOP in Theory: Some Basic Principles

Computer programs are characterized by two essential features [82]: (i) **algorithms** or, loosely, the "programming logic," and (ii) **data structures**, or how data are represented within the program, whether certain components are manipulable, iterable, etc. The object-oriented programming (OOP) paradigm, to which Python is particularly well-suited, treats these two features of a program as inseparable. Several thorough treatments of OOP are available, including texts that are independent of any language [83] and books that specifically focus on OOP in Python [84]. The core ideas are explored in this section and in Supplemental Chapters 15 and 16 in S1 Text.

Most scientific data have some form of inherent structure, and this serves as a starting point in understanding OOP. For instance, the time-series example mentioned above is structured as a series of ordered pairs, $(t, I(t))$, an X-ray diffraction pattern consists of a collection of intensities that are indexed by integer triples (h, k, l), and so on. In general, the intrinsic structure of scientific data cannot be easily or efficiently described using one of Python's standard data structures because those types (strings, lists, etc.) are far too simple and limited. Consider, for instance, the task of representing a protein 3D structure, where "representing" means storing all the information that one may wish to access and manipulate: AA sequence (residue types and numbers), the atoms comprising each residue, the spatial coordinates of each atom,

whether a cysteine residue is disulfide-bonded or not, the protein's function, the year the protein was discovered, a list of orthologs of known structure, and so on. What data structure might be capable of most naturally representing such an entity? A simple (generic) Python tuple or list is clearly insufficient.

For this problem, one could try to represent the protein as a single tuple, where the first element is a list of the sequence of residues, the second element is a string describing the protein's function, the third element lists orthologs, etc. Somewhere within this top-level list, the coordinates of the C_α atom of Alanine-42 might be represented as `[x,y,z]`, which is a simple list of length three. (The list is "simple" in the sense that its rank is one; the *rank* of a tuple or list is, loosely, the number of dimensions spanned by its rows, and in this case we have but one row.) In other words, our overall data-representation problem can be hierarchically decomposed into simpler sub-problems that **are** amenable to representation via Python's built-in types. While valid, such a data structure will be difficult to use: The programmer will have to recall multiple arbitrary numbers (list and sub-list indices) in order to access anything, and extensions to this approach will only make it clumsier. Additionally, there are many functions that are meaningful only in the context of proteins, not all tuples. For example, we may need to compute the solvent-accessible surface areas of all residues in all β-strands for a list of proteins, but this operation would be nonsensical for a list of Supreme Court cases. Conversely, not all tuple methods would be relevant to this protein data structure, yet a function to find Court cases that reached a 5-4 decision along party lines would accept the protein as an argument. In other words, the tuple mentioned above has no clean way to make the necessary associations. It's just a tuple.

OOP Terminology

This protein representation problem is elegantly solved via the OOP concepts of classes, objects, and methods. Briefly, an *object* is an instance of a data structure that contains members and methods. *Members* are data of potentially any type, including other objects. Unlike lists and tuples, where the elements are indexed by numbers starting from zero, the members of an object are given names, such as `yearDiscovered`. *Methods* are functions that (typically) make use of the members of the object. Methods perform operations that are related to the data in the object's members. Objects are constructed from *class* definitions, which are blocks that define what most of the methods will be for an object. The examples in the 'OOP in Practice' section will help clarify this terminology. (Note that some languages require that all methods and members be specified in the class declaration, but Python allows *duck punching*, or adding members after declaring a class. Adding methods later is possible too, but uncommon. Some built-in types, such as `int`, do not support duck punching.)

During execution of an actual program, a specific object is created by calling the name of the class, as one would do for a function. The interpreter will set aside some memory for the object's methods and members, and then call a method named `__init__`, which initializes the object for use.

Classes can be created from previously defined classes. In such cases, all properties of the parent class are said to be *inherited* by the child class. The child class is termed a *derived class*, while the parent is described as a *base class*. For instance, a user-defined `Biopolymer` class may have derived classes named `Protein` and `NucleicAcid`, and may itself be derived from a more general `Molecule` base class. Class names often begin with a capital letter, while object names (i.e., variables) often start with a lowercase letter. Within a class definition, a leading underscore denotes member names that will be protected. Working examples and annotated descriptions of these concepts can be found, in the context of protein structural analysis, in ref [85].

The OOP paradigm suffuses the Python language: Every value is an object. For example, the statement `foo = 'bar'` instantiates a new object (of type `str`) and binds the name `foo` to that object. All built-in string methods will be exposed for that object (e.g., `foo.upper()` returns `'BAR'`). Python's built-in `dir()` function can be used to list all attributes and methods of an object, so `dir(foo)` will list all available attributes and valid methods on the variable `foo`. The statement `dir(1)` will show all the methods and members of an `int` (there are many!). This example also illustrates the conventional OOP dot-notation, `object.attribute`, which is used to access an object's members, and to invoke its methods (Fig 1, left). For instance, `protein1.residues[2].CA.x` might give the x-coordinate of the C_α atom of the third residue in `protein1` as a floating-point number, and `protein1.residues[5].ssbond(protein2.residues[6])` might be used to define a disulfide bond (the `ssbond()` method) between residue-6 of `protein1` and residue-7 of `protein2`. In this example, the `residues` member is a list or tuple of objects, and an item is retrieved from the collection using an index in brackets.

Benefits of OOP

By effectively compartmentalizing the programming logic and implicitly requiring a disciplined approach to data structures, the OOP paradigm offers several benefits. Chief among these are (i) clean data/code separation and bundling (i.e., modularization), (ii) code reusability, (iii) greater extensibility (derived classes can be created as needs become more specialized), and (iv) encapsulation into classes/objects provides a clearer interface for other programmers and users. Indeed, a generally good practice is to discourage end-users from directly accessing and modifying all of the members of an object. Instead, one can expose a limited and clean interface to the user, while the back-end functionality (which defines the class) remains safely under the control of the class' author. As an example, custom *getter* and *setter* methods can be specified in the class definition itself, and these methods can be called in another user's code in order to enable the safe and controlled access/modification of the object's members. A setter can 'sanity-check' its input to verify that the values do not send the object into a nonsensical or broken state; e.g., specifying the string `"ham"` as the x-coordinate of an atom could be caught before program execution continues with a corrupted object. By forcing alterations and other interactions with an object to occur via a limited number of well-defined getters/setters, one can ensure that the integrity of the object's data structure is preserved for downstream usage.

The OOP paradigm also solves the aforementioned problem wherein a protein implemented as a tuple had no good way to be associated with the appropriate functions—we could call Python's built-in `max()` on a protein, which would be meaningless, or we could try to compute the isoelectric point of an arbitrary list (of Supreme Court cases), which would be similarly nonsensical. Using classes sidesteps these problems. If our `Protein` class does not define a `max()` method, then no attempt can be made to calculate its maximum. If it does define an `isoelectricPoint()` method, then that method can be applied only to an object of type `Protein`. For users/programmers, this is invaluable: If a class from a library has a particular method, one can be assured that that method will work with objects of that class.

OOP in Practice: Some Examples

A classic example of a data structure that is naturally implemented via OOP is the creation of a `Human` class. Each `Human` object can be fully characterized by her respective properties (members such as height, weight, etc.) and functionality (methods such as breathing, eating, speaking, etc.). A specific human being, e.g. `guidoVanRossum`, is an instance of the `Human` class; this class may, itself, be a subclass of a `Hominidae` base class. The following code illustrates how

one might define a Human class, including some functionality to age the Human and to set/get various members (descriptors such as height, age, etc.):

```
1 from random import randint
2
3 class Human():
4   _age = 0
5   _height = 0
6   _sex = ""
7
8   def __init__(self, theSex):
9     self.setSex(theSex)
10   def haveBirthday(self):
11     self._age += 1
12     if (self._age < 21):
13       self._height += randint(0.15)
14     if (self._age > 60):
15       self._height -= randint(0.05)
16   def setAge(self,theAge):
17     assert(theAge >= 0)
18     self._age = theAge
19   def setHeight(self,theHeight):
20     self._height = theHeight
21   def setSex(self,theSex):
22     assert(theSex == "male" or theSex == "female") # Validate input.
23     self._sex = theSex
24   def getAge(self):
25     return self._age
26   def getHeight(self):
27     return self._height
28   def getSex(self):
29     return self._sex
30 # now use the class by instantiating an object and manipulating it:
31 guido = Human("male")     # This will call __init__ and set guido's
                            # sex to "male".
32 print(guido.getAge())    # see the default value of age.
 0
33 guido.setAge(21)         # set guido's age.
34 guido.haveBirthday()     # it's guido's birthday.
35 print(guido.getAge())    # verify that age has advanced by one unit.
 22
```

Note the usage of self as the first argument in each method defined in the above code. The self keyword is necessary because when a method is invoked it must know **which** object to use. That is, an object instantiated from a class requires that methods on that object have some way to reference that particular instance of the class, versus other potential instances of that class. The self keyword provides such a "hook" to reference the specific object for which a method is called. Every method invocation for a given object, including even the initializer called __init__, must pass it**self** (the current instance) as the first argument to the method; this subtlety is further described at [86] and [87]. A practical way to view the effect of self is that any occurrence of objName.methodName(arg1, arg2) effectively becomes methodName(objName, arg1, arg2). This is one key deviation from the behavior of top-level functions, which exist outside of any class. When defining methods, usage of self provides an explicit way for the object itself to be provided as an argument (self-reference), and its disciplined usage will help minimize confusion about expected arguments.

To illustrate how objects may interact with one another, consider a class to represent a chemical's atom:

```
class Atom:
  (x,y,z) = (0,0,0) # compactly set all three vars simultaneously (rather
                    # than on three separate lines, as x = 0, y = 0, z = 0)
  name = "X"
  def distanceFrom(self, other):
    return ((self.x - other.x)**2 +
            (self.y - other.y)**2 +
            (self.z - other.z)**2)**0.5
  def setName(self, newName):
    assert(newName in ("C", "H", "O", "N", "S"))
    self.name = newName
  def moveTo(self, newX, newY, newZ):
    self.x = newX
    self.y = newY
    self.z = newZ
```

Then, we can use this `Atom` class in constructing another class to represent molecules:

```
class Molecule:
  atoms = [ ]
  bonds = [ ]
  def addAtom(self, newAtom):   # newAtom will be an Atom object
    self.atoms.append(newAtom)
  def makeBond(self, a1, a2):
    self.bonds.append((a1, a2))
  def avgBondLength(self):
    totLength = 0
    for bondIndex in range(0,len(self.bonds)):
      firstAtom = self.bonds[bondIndex][0]
      secondAtom = self.bonds[bondIndex][1]
      totLength += firstAtom.distanceFrom(secondAtom)
    return totLength / len(self.bonds)
```

And, finally, the following code illustrates the construction of a diatomic molecule:

```
a1 = Atom(); a2 = Atom()   # instantiate two atoms
a2.moveTo(1,1,0)           # move the second atom to (1,1,0)
m = Molecule()             # instantiate a new molecule
m.addAtom(a1)              # ... and populate it with
m.addAtom(a2)              #   these two atoms
m.makeBond(a1,a2)          # define a bond between the two atoms
print (m.avgBondLength())
1.4142135623730951
```

If the above code is run, for example, in an interactive Python session, then note that the aforementioned `dir()` function is an especially useful built-in tool for querying the properties of new classes and objects. For instance, issuing the statement `dir(Molecule)` will return detailed information about the `Molecule` class (including its available methods).

Exercise 9: Amino acids can be effectively represented via OOP because each AA has a well-defined chemical composition: a specific number of atoms of various element types (carbon, nitrogen, etc.) and a covalent bond connectivity that adheres to a specific pattern. For these reasons, the prototype of an L-amino acid can be unambiguously defined by the SMILES [88] string N[C@@H](R)C(= O)O, where R denotes the side-chain and @@ indicates the L enantiomer. In

addition to chemical structure, each AA also features specific physicochemical properties (molar mass, isoelectric point, optical activity/specific rotation, etc.). In this exercise, create an AA class and use it to define any two of the twenty standard AAs, in terms of their chemical composition and unique physical properties. To extend this exercise, consider expanding your AA class to include additional class members (e.g., the frequency of occurrence of that AA type) and methods (e.g., the possibility of applying post-translational modifications). To see the utility of this exercise in a broader OOP schema, see the discussion of the hierarchical **S**tructure ⊃ **M**odel ⊃ **C**hain ⊃ **R**esidue ⊃ **A**tom (SMCRA) design used in ref [85] to create classes that can represent entire protein assemblies.

File Management and I/O

Scientific data are typically acquired, processed, stored, exchanged, and archived as computer files. As a means of input/output (I/O) communication, Python provides tools for reading, writing and otherwise manipulating files in various formats. Supplemental Chapter 11 in S1 Text focuses on file I/O in Python. Most simply, the Python interpreter allows command-line input and basic data output via the `print()` function. For real-time interaction with Python, the free IPython [89] system offers a shell that is both easy to use and uniquely powerful (e.g., it features tab completion and command history scrolling); see the S2 Text, §3 for more on interacting with Python. A more general approach to I/O, and a more robust (persistent) approach to data archival and exchange, is to use files for reading, writing, and processing data. Python handles file I/O via the creation of `file` objects, which are instantiated by calling the `open` function with the filename and access mode as its two arguments. The syntax is illustrated by `fileObject = open("myName.pdb", mode = 'r')`, which creates a new file object from a file named `"myName.pdb"`. This file will be only readable because the `'r'` mode is specified; other valid modes include `'w'` to allow writing and `'a'` for appending. Depending on which mode is specified, different methods of the file object will be exposed for use. Table 4 describes mode types and the various methods of a `File` object.

The following example opens a file named `myDataFile.txt` and reads the lines, *en masse*, into a list named `listOfLines`. (In this example, the variable `readFile` is also known as a "file handle," as it references the file object.) As for all lists, this object is iterable and can be looped over in order to process the data.

```
readFile = open("myDataFile.txt", mode = 'r')
listOfLines = readFile.readlines()
# Process the lines. Simply dump the contents to the console:
for l in listOfLines:
  print(l)
 (The lines in the file will be printed)
readFile.close()
```

Table 4. Python's file-access modes.

I/O Mode	Syntax	Behavior
Read	'r'	Opens the contents of a file for reading into the file interface, allowing for lines to be read-in successively.
Write	'w'	Creates a file with the specified name and allows for text to be written to the file; note that specifying a pre-existing filename will overwrite the existing file.
Append	'a'	Opens an existing file and allows for text to be written to it, starting at the conclusion of the original file contents.
Read and Write	'r+'	Opens a file such that its contents can be both read-in and written-to, thus offering great versatility.

Python's available file-access modes are summarized here.

doi:10.1371/journal.pcbi.1004867.t004

Data can be extracted and processed via subsequent string operations on the list of lines drawn from the file. In fact, many data-analysis workflows commit much effort to the pre-processing of raw data and standardization of formats, simply to enable data structures to be cleanly populated. For many common input formats such as .csv (comma-separated values) and .xls (Microsoft Excel), packages such as pandas [90] simplify the process of reading in complex file formats and organizing the input as flexible data structures. For more specialized file formats, much of this 'data wrangling' stems from the different degrees of standards-compliance of various data sources, as well as the immense heterogeneity of modern collections of datasets (sequences, 3D structures, microarray data, network graphs, etc.). A common example of the need to read and extract information is provided by the PDB file format [22], which is a container for macromolecular structural data. In addition to its basic information content—lists of atoms and their 3D coordinates—the standard PDB file format also includes a host of *metadata* (loosely, data that describe other (lower-level) data, for instance in terms of syntax and schemas), such as the biopolymer sequence, protein superfamily, quaternary structures, chemical moieties that may be present, X-ray or NMR refinement details, and so on. Indeed, processing and analyzing the rich data available in a PDB file motivates the Final Project at the end of this primer. For now, this brief example demonstrates how to use Python's I/O methods to count the number of HETATM records in a PDB file:

```
1 fp = open('1I8F.pdb', mode = 'r')
2 numHetatm = 0
3 for line in fp.readlines():
4   if(len(line) > 6):
5     if(line[0:6] == "HETATM"):
6       numHetatm += 1
7 fp.close()
8 print(numHetatm)
  160
```

Such HETATM, or heteroatom, lines in a PDB file correspond to water, ions, small-molecule ligands, and other non-biopolymer components of a structure; for example, glycerol HETATM lines are often found in cryo-crystallographic structures, where glycerol was added to crystals as a cryo-protectant.

Exercise 10: The standard FASTA file-format, used to represent protein and nucleic acid sequences, consists of two parts: (i) The first line is a description of the biomolecule, starting with a greater-than sign (>) in the first column; this sign is immediately followed by a non-whitespace character and any arbitrary text that describes the sequence name and other information (e.g., database accession identifiers). (ii) The subsequent lines specify the biomolecular sequence as single-letter codes, with no blank lines allowed. A protein example follows:

```
>tr|Q8ZYG5|Q8ZYG5_PYRAE (Sm-like) OS = P aerophilum GN = PAE0790
MASDISKCFATLGATLQDSIGKQVLVKLRDSHEIRGILRSFDQHVNLLLEDAEEIIDGNV
YKRGTMVVRGENVLFISPVP
```

Begin this exercise by choosing a FASTA protein sequence with more than 3000 AA residues. Then, write Python code to read in the sequence from the FASTA file and: (i) determine the relative frequencies of AAs that follow proline in the sequence; (ii) compare the distribution of AAs that follow proline to the distribution of AAs in the entire protein; and (iii) write these results to a human-readable file.

Regular Expressions for String Manipulations

The *regular expression* (*regex*) is an extensible tool for pattern matching in strings. They are discussed at length in Supplemental Chapter 17 in S1 Text. Regexes entered the world of

practical programming in the late 1960s at Bell Labs and, like many tools of that era, they are powerful, flexible, and terse constructs. Fundamentally, a regex specifies a set of strings. The simplest type of regex is a simple string with no special characters (*metacharacters*). Such a regex will match itself: `Biology` would match "Biology" or "Biologys," but not "biology," "Biochem," or anything else that does not start with "Biology" (note the case sensitivity).

In Python, a regex `match`es a string if the string starts with that regex. Python also provides a `search` function to locate a regex anywhere within a string. Returning to the notion that a regex "specifies a set of strings," given some text the `match`es to a regex will be all strings that **start** with the regex, while the `search` hits will be all strings that **contain** the regex. For clarity, we will say that a regex `find`s a string if the string is completely described by the regex, with no trailing characters. (There is no `find` in Python but, for purposes of description here, it is useful to have a term to refer to a `match` without trailing characters.)

Locating strings and parsing text files is a ubiquitous task in the biosciences, e.g. identifying a stop codon in a nucleic acid FASTA file or finding error messages in an instrument's log files. Yet regexes offer even greater functionality than may be initially apparent from these examples, as described below. First, we note that the following metacharacters are special in regexes: `$^.*+?{}[]()|\`, and in most cases they do not `find` themselves.

The `^` and `$` metacharacters (known as *anchors*) are straightforward, as they `find` the start and end of a line, respectively. While `match` looks for lines beginning with the specified regex, adding a `$` to the end of the regex pattern will ensure that any matching line ends at the end of the regex. (This is why there is no `find` function in Python: it is easily achieved by adding a `$` to a regex used in `match`.) For example, to `find` lines in a log file that state 'Run complete', but not 'Run completes in 5 minutes', the regex `Run complete$` would `match` the desired target lines.

A `.` (a period) `find`s literally any character. For example, if a protein kinase has a consensus motif of 'AXRSXRSXRSP', where X is any AA, then the regex `A.RS.RS.RSP` would succeed in `search`ing for substrates.

The metacharacters `*`, `+`, `{}`, and `?` are special *quantifier* operators, used to specify repetition of a character, character class, or higher-order unit within a regex (described below). A `*` after a character (or group of characters) `find`s that character zero or more times. Returning to the notion of a consensus motif, a protein that recognizes RNA which contains the dinucleotide 'UG' followed by any number of 'A's would find its binding partners by `search`ing for the regex `UGA*`. One can comb through RNA-seq reads to find sequences that are 3′-polyadenylated by `search`ing for `AAAAAA*$`. This would `find` exactly five 'A's, followed by zero or more 'A's, followed by the end of the line. The `+` metacharacter is akin to `*`, except that it `find`s one or more of the preceding character. A `?` `find`s the preceding character zero or one time. Most generally, the `{m, n}` syntax `find`s the preceding character (possibly from a character class) between *m* and *n* times, inclusive. Thus, `x{3}` `find`s the character 'x' if repeated exactly three times, `A{5,18}` `find`s the character 'A' repeated five to eighteen times, and `P{2,}` `find`s runs of two or more 'P' characters.

Combining the above concepts, we can `search` for protein sequences that begin with a His6×-tag ('HHHHHH'), followed by at most five residues, then a TEV protease cleavage site ('ENLYFQ'), followed immediately by a 73-residue polypeptide that ends with 'IIDGNV'. The regex to `search` for this sequence would be `H{6}.{0,5}ENLYFQ.{67}IIDGNV`.

Characters enclosed in square brackets, `[]`, specify a *character class*. This functionality allows a regex to `find` any of a set of characters. For example, `AAG[TC]G` would `find` 'AAG**T**G' or 'AAG**C**G', where the variable char from the character class is bolded. A range of characters can be provided by separating them with a hyphen, `-`. So, for instance, `[A-Z][a-z]*` would `find` a

word that starts with a capital letter. Multiple ranges can be specified, and `[1-9][A-Za-z0-9]{3}.pdb` would find PDB files in some search directory. (Note that the `.` in that regex will find any character, so '1I8Fnpdb' would be matched, even though we might intend for only '1I8F.pdb' to be found. This could be corrected by escaping the `.` with a backslash, as discussed below.) The `^` metacharacter can be used to negate a character class: `[^0-9]` would find any non-numeric character.

The backslash metacharacter, `\`, is used to suppress, or *escape*, the meaning of the immediately following character; for this reason, `\` is known as an *escape character*. For example, consider the task of finding prices exceeding $1000 in a segment of text. A regex might be `$0*[1-9][0-9]{3,}.[0-9]{2}`. This monstrous regex should find a dollar sign, any number of zeros, one non-zero number, at least three numbers, a period, and two numbers. Thus, '$01325.25' would be found, but not '$00125.67'. (The requirement of a non-zero number followed by three numbers is not met in this case.) But, there is a problem here: The `$` metacharacter anchors the end of a line, and because no text can appear after the end of a line this regex will never match any text. Furthermore, the `.` is meant to find a literal period (the decimal point), but in a regex it is a wildcard that finds any character. The `\` metacharacter can be used to solve both of these problems: It notifies the regex engine to treat the subsequent character as a literal. Thus, a correct regex for prices over $1000 would be `\$0*[1-9][0-9]{3,}\.[0-9]{2}`. To find a literal '\', use `\\`. (The `\` metacharacter often appears in I/O processing as a way to escape quotation marks; for instance, the statement `print("foo")` will output `foo`, whereas `print("\"foo\"")` will print `"foo"`.)

Python and many other languages include a `\` before certain (non-reserved) characters, as a convenient built-in feature for commonly-used character classes. In particular, `\d` finds any digit, `\s` finds whitespace, `\S` finds non-whitespace, and `\w` finds any alphanumeric character or underscore (i.e., the class `[a-zA-Z0-9_]`), such as typically occurs in ordinary English words. These built-in features can be used to more compactly express the price regex, including the possibility of whitespace between the '$' sign and the first digit: `\$\s*0*[1-9]\d{3,}\.\d{2}`.

The `|` metacharacter is the logical 'or' operator (also known as the *alternation* operator). The regex `abc|xyz` will find either 'abc' or 'xyz'. Initially, the behavior of `|` can be deceptive: `£|€|$ [0-9]*` is not equivalent to `[£€$] [0-9]*`, as the former will find a lone pound symbol, a lone Euro symbol, or a dollar sign followed by a number. As an example, to match the SEQRES and ATOM records in a PDB file, `ATOM.*|SEQRES.*` would work.

The final metacharacters that we will explore are matched parentheses, `()`, which find *character groups*. While `x[abc]y` will find 'xay', 'xby', or 'xcy', the regex `x(abc)y` matches only those strings starting with 'xabcy'—i.e., it is equivalent to `xabcy`. The utility of groups stems from the ability to use them as units of repetition. For example, to see if a sequence is delimited by a start and stop codon, and therefore is a potential ORF, we could use `AUG.*U(AA|AG|GA)`; this regex will search for 'UAA', 'UAG', or 'UGA' at the end of the sequence. (Note that parentheses delimit the `|`.) Note that this regex does not check that the start and stop codon are in the same frame, since the characters that find captures by the `.*` may not be a multiple of three. To address this, the regex could be changed to `AUG(...)*U(AA|AG|GA)`. Another feature of groups is the ability to refer to previous occurrences of a group within the regex (a *backreference*), enabling even more versatile pattern matching. To explore groups and other powerful features of regexes, readers can consult thorough texts [91] and numerous online resources (e.g., [92,93]).

Beyond the central role of the regex in analyzing biological sequences, parsing datasets, etc., note that any effort spent learning Python regexes is highly transferable. In terms of general

syntactic forms and functionality, regexes behave roughly similarly in Python and in many other mainstream languages (e.g., Perl, R), as well as in the shell scripts and command-line utilities (e.g., grep) found in the Unix family of operating systems (including all Linux distributions and Apple's OS X).

Exercise 11: Many human hereditary neurodegenerative disorders, such as Huntington's disease (HD), are linked to anomalous expansions in the number of trinucleotide repeats in particular genes [94]. In HD, the pathological severity correlates with the number of $(CAG)_n$ repeats in exon-1 of the gene (*htt*) encoding the protein (huntingtin): More repeats means an earlier age of onset and a more rapid disease progression. The CAG codon specifies glutamine, and HD belongs to a broad class of polyglutamine (polyQ) diseases. Healthy (wild-type) variants of this gene feature $n \approx 6$–35 tandem repeats, whereas $n > 35$ virtually assures the disease. For this exercise, write a Python regex that will locate any consecutive runs of $(CAG)_{n>10}$ in an input DNA sequence. Because the codon CAA also encodes Q and has been found in long runs of CAGs, your regex should also allow interspersed CAAs. To extend this exercise, write code that uses your regex to count the number of CAG repeats (allow CAA too), and apply it to a publically-available genome sequence of your choosing (e.g., the NCBI GI code 588282786:1-585 is exon-1 from a human's *htt* gene [accessible at http://1.usa.gov/1NjrDNJ]).

An Advanced Vignette: Creating Graphical User Interfaces with Tkinter

Thus far, this primer has centered on Python programming as a tool for interacting with data and processing information. To illustrate an advanced topic, this section shifts the focus towards approaches for creating software that relies on user interaction, via the development of a graphical user interface (GUI; pronounced 'gooey'). Text-based interfaces (e.g., the Python shell) have several distinct advantages over purely graphical interfaces, but such interfaces can be intimidating to the uninitiated. For this reason, many general users will prefer GUI-based software that permits options to be configured via graphical check boxes, radio buttons, pull-down menus and the like, versus text-based software that requires typing commands and editing configuration files. In Python, the `tkinter` package (pronounced 'T-K-inter') provides a set of tools to create GUIs. (Python 2.x calls this package `Tkinter`, with a capital `T`; here, we use the Python 3.x notation.)

Tkinter programming has its own specialized vocabulary. *Widgets* are objects, such as text boxes, buttons and frames, that comprise the user interface. The *root window* is the widget that contains all other widgets. The root window is responsible for monitoring user interactions and informing the contained widgets to respond when the user triggers an interaction with them (called an *event*). A *frame* is a widget that contains other widgets. Frames are used to group related widgets together, both in the code and on-screen. A *geometry manager* is a system that places widgets in a frame according to some style determined by the programmer. For example, the `grid` geometry manager arranges widgets on a grid, while the `pack` geometry manager places widgets in unoccupied space. Geometry managers are discussed at length in Supplemental Chapter 18 in S1 Text, which shows how intricate layouts can be generated.

The basic style of GUI programming fundamentally differs from the material presented thus far. The reason for this is that the programmer cannot predict what actions a user might perform, and, more importantly, in what order those actions will occur. As a result, GUI programming consists of placing a set of widgets on the screen and providing instructions that the widgets execute when a user interaction triggers an event. (Similar techniques are used, for instance, to create web interfaces and widgets in languages such as JavaScript.) Supplemental Chapter 19 (S1 Text) describes available techniques for providing functionality to widgets.

Once the widgets are configured, the root window then awaits user input. A simple example follows:

```
1 from tkinter import Tk, Button
2 def buttonWindow():
3   window = Tk()
4   def onClick():
5     print("Button clicked")
6   btn = Button(window, text = "Sample Button", command = onClick)
7   btn.pack()
8   window.mainloop()
```

To spawn the Tk window, enter the above code in a Python shell and then issue the statement `buttonWindow()`. Then, press the "Sample Button" while viewing the output on the console. The first line in the above code imports the `Tk` and `Button` classes. `Tk` will form the root window, and `Button` will create a button widget. Inside the function, line 3 creates the root window. Lines 4 and 5 define a function that the button will call when the user interacts with it. Line 6 creates the button. The first argument to the `Button` constructor is the widget that will contain the button, and in this case the button is placed directly in the root window. The `text` argument specifies the text to be displayed on the button widget. The `command` argument attaches the function named `onClick` to the button. When the user presses the button, the root window will instruct the button widget to call this function. Line 7 uses the `pack` geometry manager to place the button in the root window. Finally, line 8 instructs the root window to enter `mainloop`, and the root window is said to *listen* for user input until the window is closed.

Graphical widgets, such as text entry fields and check-boxes, receive data from the user, and must communicate that data within the program. To provide a conduit for this information, the programmer must provide a variable to the widget. When the value in the widget changes, the widget will update the variable and the program can read it. Conversely, when the program should change the data in a widget (e.g., to indicate the status of a real-time calculation), the programmer sets the value of the variable and the variable updates the value displayed on the widget. This roundabout tack is a result of differences in the architecture of Python and Tkinter—an integer in Python is represented differently than an integer in Tkinter, so reading the widget's value directly would result in a nonsensical Python value. These variables are discussed in Supplemental Chapter 19 in S1 Text.

From a software engineering perspective, a drawback to graphical interfaces is that multiple GUIs cannot be readily composed into new programs. For instance, a GUI to display how a particular restriction enzyme will cleave a DNA sequence will not be practically useful in predicting the products of digesting thousands of sequences with the enzyme, even though some core component of the program (the key, non-GUI program logic) would be useful in automating that task. For this reason, GUI applications should be written in as modular a style as possible—one should be able to extract the useful functionality without interacting with the GUI-specific code. In the restriction enzyme example, an optimal solution would be to write the code that computes cleavage sites as a separate module, and then have the GUI code interact with the components of that module.

Python in General-purpose Scientific Computing: Numerical Efficiency, Libraries

In pursuing biological research, the computational tasks that arise will likely resemble problems that have already been solved, problems for which software libraries already exist. This

occurs largely because of the interdisciplinary nature of biological research, wherein relatively well-established formalisms and algorithms from physics, computer science, and mathematics are applied to biological systems. For instance, (i) the simulated annealing method was developed as a physically-inspired approach to combinatorial optimization, and soon thereafter became a cornerstone in the refinement of biomolecular structures determined by NMR spectroscopy or X-ray crystallography [95]; (ii) dynamic programming was devised as an optimization approach in operations research, before becoming ubiquitous in sequence alignment algorithms and other areas of bioinformatics; and (iii) the Monte Carlo method, invented as a sampling approach in physics, underlies the algorithms used in problems ranging from protein structure prediction to phylogenetic tree estimation.

Each computational approach listed above can be implemented in Python. The language is well-suited to rapidly develop and prototype any algorithm, be it intended for a relatively lightweight problem or one that is more computationally intensive (see [96] for a text on general-purpose scientific computing in Python). When considering Python and other possible languages for a project, software development time must be balanced against a program's execution time. These two factors are generally countervailing because of the inherent performance trade-offs between codes that are written in interpreted (high-level) versus compiled (lower-level) languages; ultimately, the computational demands of a problem will help guide the choice of language. In practice, the feasibility of a pure Python versus non-Python approach can be practically explored via numerical benchmarking. While Python enables rapid development, and is of sufficient computational speed for many bioinformatics problems, its performance simply cannot match the compiled languages that are traditionally used for high-performance computing applications (e.g., many MD integrators are written in C or Fortran). Nevertheless, Python codes are available for molecular simulations, parallel execution, and so on. Python's popularity and utility in the biosciences can be attributed to its ease of use (expressiveness), its adequate numerical efficiency for many bioinformatics calculations, and the availability of numerous libraries that can be readily integrated into one's Python code (and, conversely, one's Python code can "hook" into the APIs of larger software tools, such as PyMOL). Finally, note that rapidly-developed Python software can be integrated with numerically efficient, high-performance code written in a low-level languages such as C, in an approach known as "mixed-language programming" [49].

Many third-party Python libraries are now well-established. In general, these mature projects are (i) well-documented, (ii) freely available as stable (production) releases, (iii) undergoing continual development to add new features, and (iv) characterized by large user-bases and active communities (mailing lists, etc.). A useful collection of such tools can be found at the SciPy resource [97,98], which is a platform for the maintenance and distribution of several popular packages: (i) NumPy, which is invaluable for matrix-related calculations [99]; (ii) SciPy, which provides routines from linear algebra, signal processing, statistics, and a wealth of other numerical tools; (iii) pandas, which facilitates data import, management, and organization [90]; and (iv) matplotlib, a premiere codebase for plotting and general-purpose visualization [62]. The package scikit-learn extends SciPy with machine learning and statistical analysis functionalities [100]. Other statistical tools are available in the statistics standard library, in SciPy [97,98], and in NumPy [99]; finally, many more-specialized packages also exist, such as pyBrain [78] and DEAP [101]. Properly interacting with Python modules, such as those mentioned above, is detailed in Supplemental Chapter 4 (S1 Text).

Many additional libraries can be found at the official Python Package Index (PyPI; [102]), as well as myriad packages from unofficial third-party repositories. The BioPython project, mentioned above in the 'Why Python?' subsection, offers an integrated suite of tools for

sequence- and structure-based bioinformatics, as well as phylogenetics, machine learning, and other feature sets. We survey the computational biology software landscape in the S2 Text (§2), including tools for structural bioinformatics, phylogenetics, omics-scale data-processing pipelines, and workflow management systems. Finally, note that Python code can be interfaced with other languages. For instance, current support is provided for low-level integration of Python and R [103,104], as well as C-extensions in Python (Cython; [105,106]). Such cross-language interfaces extend Python's versatility and flexibility for computational problems at the intersection of multiple scientific domains, as often occurs in the biosciences.

Python and Software Licensing

Any discussion of libraries, modules, and extensions merits a brief note on the important role of licenses in scientific software development. As evidenced by the widespread utility of existing software libraries in modern research communities, the development work done by one scientist will almost certainly aid the research pursuits of others—either near-term or long-term, in subfields that might be near to one's own or perhaps more distant (and unforeseen). *Free software* licenses promote the unfettered advance of scientific research by encouraging the open exchange, transparency, communicability, and reproducibility of research projects. To qualify as free software, a program must allow the user to view and change the source code (for any purpose), distribute the code to others, and distribute modified versions of the code to others. The Open Source Initiative provides alphabetized and categorized lists of licenses that comply, to various degrees, with the open-source definition [107]. As an example, the Python interpreter, itself, is under a free license. Software licensing is a major topic unto itself, and helpful primers are available on technical [38] and strategic [37,108] considerations in adopting one licensing scheme versus another. All of the content (code and comments) that is provided as Supplemental Chapters (S1 Text) is licensed under the GNU Affero General Public License (AGPL) version 3, which permits anyone to examine, edit, and distribute the source so long as any works using it are released under the same license.

Managing Large Projects: Version Control Systems

As a project grows, it becomes increasingly difficult—yet increasingly important—to be able to track changes in source code. A *version control system* (VCS) tracks changes to documents and facilitates the sharing of code among multiple individuals. In a *distributed* (as opposed to centralized) VCS, each developer has his own complete copy of the project, locally stored. Such a VCS supports the "committing," "pulling," "branching," and "merging" of code. After making a change, the programmer *commits* the change to the VCS. The VCS stores a snapshot of the project, preserving the development history. If it is later discovered that a particular commit introduced a bug, one can easily revert the offending commit. Other developers who are working on the same project can *pull* from the author of the change (the most recent version, or any earlier snapshot). The VCS will incorporate the changes made by the author into the puller's copy of the project. If a new feature will make the code temporarily unusable (until the feature is completely implemented), then that feature should be developed in a separate *branch*. Developers can switch between branches at will, and a commit made to one branch will not affect other branches. The *master branch* will still contain a working version of the program, and developers can still commit non-breaking changes to the master branch. Once the new feature is complete, the branches can be *merged* together. In a distributed VCS, each developer is, conceptually, a branch. When one developer pulls from others, this is equivalent to merging a branch from each developer. Git, Mercurial, and Darcs are common distributed VCS. In contrast, in a *centralized* VCS all commits are tracked in one central place (for both distributed

and centralized VCS, this "place" is often a repository hosted in the cloud). When a developer makes a commit, it is pushed to every other developer (who is on the same branch). The essential behaviors—committing, branching, merging—are otherwise the same as for a distributed VCS. Examples of popular centralized VCSs include the Concurrent Versioning System (CVS) and Subversion.

While VCS are mainly designed to work with source code, they are not limited to this type of file. A VCS is useful in many situations where multiple people are collaborating on a single project, as it simplifies the task of combining, tracking, and otherwise reconciling the contributions of each person. In fact, this very document was developed using LaTeX and the Git VCS, enabling each author to work on the text in parallel. A helpful guide to Git and GitHub (a popular Git repository hosting service) was very recently published [109]; in addition to a general introduction to VCS, that guide offers extensive practical advice, such as what types of data/files are more or less ideal for version controlling.

Final Project: A Structural Bioinformatics Problem

Fluency in a programming language is developed actively, not passively. The exercises provided in this text have aimed to develop the reader's command of basic features of the Python language. Most of these topics are covered more deeply in the Supplemental Chapters (S1 Text), which also include some advanced features of the language that lie beyond the scope of the main body of this primer. As a final exercise, a cumulative project is presented below. This project addresses a substantive scientific question, and its successful completion requires one to apply and integrate the skills from the foregoing exercises. Note that a project such as this—and really any project involving more than a few dozen lines of code—will benefit greatly from an initial planning phase. In this initial stage of software design, one should consider the basic functions, classes, algorithms, control flow, and overall code structure.

Exercise 12 (cumulative project): First, obtain a set of several hundred protein structures from the PDB, as plaintext .pdb files (the exact number of entries is immaterial). Then, from this pool of data, determine the relative frequencies of the constituent amino acids for each protein secondary structural class; use only the three descriptors "helix," "sheet," and, for any AA not within a helix or sheet, "irregular." (Hint: In considering file parsing and potential data structures, search online for the PDB's file-format specifications.) Output your statistical data to a human-readable file format (e.g., comma-separated values, .csv) such that the results can be opened in a statistical or graphical software package for further processing and analysis. As a bonus exercise, use Python's matplotlib package to visualize the findings of your structural bioinformatics analysis.

Conclusion

Data and algorithms are two pillars of modern biosciences. Data are acquired, filtered, and otherwise manipulated in preparation for further processing, and algorithms are applied in analyzing datasets so as to obtain results. In this way, computational workflows transform primary data into results that can, over time, become formulated into general principles and new knowledge. In the biosciences, modern scientific datasets are voluminous and heterogeneous. Thus, in developing and applying computational tools for data analysis, the two central goals are **scalability**, for handling the data-volume problem, and **robust abstractions**, for handling data heterogeneity and integration. These two challenges are particularly vexing in biology, and are exacerbated by the traditional lack of training in computational and quantitative methods in many biosciences curricula. Motivated by these factors, this primer has sought to introduce general principles of computer programming, at both basic and intermediate levels. The

Python language was adopted for this purpose because of its broad prevalence and deep utility in the biosciences.

Supporting Information

S1 Text. Python Chapters. This suite of 19 Supplemental Chapters covers the essentials of programming. The Chapters are written in Python and guide the reader through the core concepts of programming, via numerous examples and explanations. The most recent versions of all materials are maintained at http://p4b.muralab.org. For purposes of self-study, solutions to the in-text exercises are also included.
(ZIP)

S2 Text. Supplemental text. The supplemental text contains sections on: *(i)* Python as a general language for scientific computing, including the concepts of imperative and declarative languages, Python's relationship to other languages, and a brief account of languages widely used in the biosciences; *(ii)* a structured guide to some of the available software packages in computational biology, with an emphasis on Python; and *(iii)* two sample Supplemental Chapters (one basic, one more advanced), along with a brief, practical introduction to the Python interpreter and integrated development environment (IDE) tools such as IDLE.
(PDF)

Acknowledgments

We thank M. Cline, S. Coupe, S. Ehsan, D. Evans, R. Sood, and K. Stanek for critical reading and helpful feedback on the manuscript.

References

1. Metzker ML. Sequencing Technologies—The Next Generation. Nature Reviews: Genetics. 2010 Jan; 11(1):31–46. doi: 10.1038/nrg2626 PMID: 19997069
2. Larance M, Lamond AI. Multidimensional Proteomics for Cell Biology. Nature Reviews: Molecular Cell Biology. 2015 May; 16(5):269–280. doi: 10.1038/nrm3970 PMID: 25857810
3. Wang Z, Gerstein M, Snyder M. RNA-Seq: A Revolutionary Tool for Transcriptomics. Nature Reviews: Genetics. 2009 Jan; 10(1):57–63. doi: 10.1038/nrg2484 PMID: 19015660
4. Wishart DS. Computational Approaches to Metabolomics. Methods in Molecular Biology. 2010; 593:283–313. doi: 10.1007/978-1-60327-194-3_14 PMID: 19957155
5. OMICS: A Journal of Integrative Biology;. Available from: http://www.liebertpub.com/overview/omics-a-journal-of-integrative-biology/43.
6. Lederberg J. 'Ome Sweet 'Omics—A Genealogical Treasury of Words. The Scientist. 2001 April; Available from: http://www.the-scientist.com/?articles.view/articleNo/13313/title/-Ome-Sweet–Omics—A-Genealogical-Treasury-of-Words.
7. Gerstein Lab. "O M E S Table";. Available from: http://bioinfo.mbb.yale.edu/what-is-it/omes/omes.html.
8. Bolhuis H, Cretoiu MS, Stal LJ. Molecular Ecology of Microbial Mats. FEMS Microbiology Ecology. 2014 Nov; 90(2):335–350. PMID: 25109247
9. Tottey W, Denonfoux J, Jaziri F, Parisot N, Missaoui M, Hill D, et al. The Human Gut Chip "HuGChip", an Explorative Phylogenetic Microarray for Determining Gut Microbiome Diversity at Family Level. PLoS ONE. 2013; 8(5):e62544. doi: 10.1371/journal.pone.0062544 PMID: 23690942
10. Turner TL, Bourne EC, Von Wettberg EJ, Hu TT, Nuzhdin SV. Population Resequencing Reveals Local Adaptation of *Arabidopsis lyrata* to Serpentine Soils. Nature Genetics. 2010 Mar; 42(3):260–263. doi: 10.1038/ng.515 PMID: 20101244
11. Shendure J, Ji H. Next-generation DNA Sequencing. Nature Biotechnology. 2008 Oct; 26(10):1135–1145. doi: 10.1038/nbt1486 PMID: 18846087
12. Schuster SC. Next-generation Sequencing Transforms Today's Biology. Nature Methods. 2008 Jan; 5(1):16–18. doi: 10.1038/nmeth1156 PMID: 18165802

13. Glenn TC. Field Guide to Next-generation DNA Sequencers. Molecular Ecology Resources. 2011 Sep; 11(5):759–769. doi: 10.1111/j.1755-0998.2011.03024.x PMID: 21592312

14. Klock HE, White A, Koesema E, Lesley SA. Methods and Results for Semi-automated Cloning Using Integrated Robotics. Journal of Structural and Functional Genomics. 2005; 6(2–3):89–94. doi: 10.1007/s10969-005-3084-1 PMID: 16211504

15. Viola R, Carman P, Walsh J, Frankel D, Rupp B. Automated Robotic Harvesting of Protein Crystals—Addressing a Critical Bottleneck or Instrumentation Overkill? Journal of Structural and Functional Genomics. 2007 Dec; 8(4):145–152. PMID: 17965947

16. Baldi P, Brunak S. Bioinformatics: The Machine Learning Approach (2nd Edition). A Bradford Book; 2001.

17. Mura C, McAnany CE. An Introduction to Biomolecular Simulations and Docking. Molecular Simulation. 2014 Aug; 40(10–11):732–764. doi: 10.1080/08927022.2014.935372

18. Ramanathan A, Agarwal PK, Kurnikova M, Langmead CJ. An Online Approach for Mining Collective Behaviors from Molecular Dynamics Simulations. Journal of Computational Biology. 2010 Mar; 17 (3):309–324. doi: 10.1089/cmb.2009.0167 PMID: 20377447

19. Tribello GA, Bonomi M, Branduardi D, Camilloni C, Bussi G. PLUMED 2: New Feathers for an Old Bird. Computer Physics Communications. 2014 Feb; 185(2):604–613. doi: 10.1016/j.cpc.2013.09.018

20. Cieślik M, Mura C. A Lightweight, Flow-based Toolkit for Parallel and Distributed Bioinformatics Pipelines. BMC Bioinformatics. 2011; 12:61. doi: 10.1186/1471-2105-12-61 PMID: 21352538

21. Levitt M. Nature of the Protein Universe. Proceedings of the National Academy of Sciences of the United States of America. 2009 Jul; 106(27):11079–11084. doi: 10.1073/pnas.0905029106 PMID: 19541617

22. Bernstein FC, Koetzle TF, Williams GJ, Meyer EF, Brice MD, Rodgers JR, et al. The Protein Data Bank: A Computer-based Archival File for Macromolecular Structures. Journal of Molecular Biology. 1977 May; 112(3):535–542. doi: 10.1016/S0022-2836(77)80200-3 PMID: 875032

23. Pennisi E. Human genome 10th anniversary. Will computers crash genomics? Science. 2011 Feb; 331(6018):666–668. doi: 10.1126/science.331.6018.666 PMID: 21310981

24. Barabási AL, Gulbahce N, Loscalzo J. Network Medicine: A Network-based Approach to Human Disease. Nature Reviews Genetics. 2011 Jan; 12(1):56–68. doi: 10.1038/nrg2918 PMID: 21164525

25. Lichtman JW, Pfister H, Shavit N. The Big Data Challenges of Connectomics. Nature Neuroscience. 2014 Nov; 17(11):1448–1454. doi: 10.1038/nn.3837 PMID: 25349911

26. Engert F. The Big Data Problem: Turning Maps into Knowledge. Neuron. 2014 Sep; 83(6):1246–1248. doi: 10.1016/j.neuron.2014.09.008 PMID: 25233305

27. Costa FF. Big Data in Biomedicine. Drug Discovery Today. 2014 Apr; 19(4):433–440. doi: 10.1016/j.drudis.2013.10.012 PMID: 24183925

28. Rudin C, Dunson D, Irizarry R, Ji H, Laber E, Leek J, et al. Discovery with Data: Leveraging Statistics with Computer Science to Transform Science and Society. American Statistical Association; 2014. Available from: http://www.amstat.org/policy/pdfs/BigDataStatisticsJune2014.pdf.

29. Committee on a New Biology for the 21st Century: Ensuring the United States Leads the Coming Biology Revolution and Board on Life Sciences and Division on Earth and Life Studies and National Research Council. A New Biology for the 21st Century. National Academies Press; 2009.

30. Pevzner P, Shamir R. Computing Has Changed Biology—Biology Education Must Catch Up. Science. 2009 Jul; 325(5940):541–542. doi: 10.1126/science.1173876 PMID: 19644094

31. Tan T, Lim S, Khan AM, Ranganathan S. A Proposed Minimum Skill Set for University Graduates to Meet the Informatics Needs and Challenges of the "-omics" Era. BMC Genomics. 2009; 10(Suppl 3): S36. doi: 10.1186/1471-2164-10-S3-S36 PMID: 19958501

32. Welch L, Lewitter F, Schwartz R, Brooksbank C, Radivojac P, Gaeta B, et al. Bioinformatics Curriculum Guidelines: Toward a Definition of Core Competencies. PLoS Comput Biol. 2014 Mar; 10(3): e1003496. doi: 10.1371/journal.pcbi.1003496 PMID: 24603430

33. Rubinstein A, Chor B. Computational Thinking in Life Science Education. PLoS Comput Biol. 2014 Nov; 10(11):e1003897. doi: 10.1371/journal.pcbi.1003897 PMID: 25411839

34. Goodman AL, Dekhtyar A. Teaching Bioinformatics in Concert. PLoS Comput Biol. 2014 Nov; 10(11): e1003896. doi: 10.1371/journal.pcbi.1003896 PMID: 25411792

35. McIlroy M, Pinson E, Tague B. UNIX Time-sharing System. The Bell System Technical Journal. 1978 July-August; 57(6):1899–1904. doi: 10.1002/j.1538-7305.1978.tb02135.x

36. Steele GL. Growing a Language. Higher Order Symbolic Computation. 1999 October; 12(3):221–236. For the video, see http://www.youtube.com/watch?v=_ahvzDzKdB0. doi: 10.1023/A:1010085415024

37. Barnes N. Publish Your Computer Code: It Is Good Enough. Nature. 2010 Oct; 467(7317):753–753. doi: 10.1038/467753a PMID: 20944687

38. Morin A, Urban J, Sliz P. A Quick Guide to Software Licensing for the Scientist-Programmer. PLoS Comput Biol. 2012 Jul; 8(7):e1002598. doi: 10.1371/journal.pcbi.1002598 PMID: 22844236

39. Abelson H, Sussman GJ. Structure and Interpretation of Computer Programs (2nd Edition). The MIT Press; 1996. Available from: http://mitpress.mit.edu/sicp/full-text/book/book.html.

40. Evans D. Introduction to Computing: Explorations in Language, Logic, and Machines. CreateSpace Independent Publishing Platform; 2011. Available from: http://www.computingbook.org.

41. The PyMOL Molecular Graphics System, Schrödinger, LLC;. Available from: http://pymol.org.

42. Humphrey W, Dalke A, Schulten K. VMD—Visual Molecular Dynamics. Journal of Molecular Graphics. 1996; 14:33–38. doi: 10.1016/0263-7855(96)00018-5 PMID: 8744570

43. Emsley P, Lohkamp B, Scott WG, Cowtan K. Features and Development of Coot. Acta Crystallographica Section D—Biological Crystallography. 2010; 66:486–501. doi: 10.1107/S0907444910007493

44. Mura C, McCrimmon CM, Vertrees J, Sawaya MR. An Introduction to Biomolecular Graphics. PLoS Comput Biol. 2010 Aug; 6(8):e1000918. doi: 10.1371/journal.pcbi.1000918 PMID: 20865174

45. PBCTools Plugin, Version 2.7;. Available from: http://www.ks.uiuc.edu/Research/vmd/plugins/pbctools.

46. Fourment M, Gillings MR. A Comparison of Common Programming Languages Used in Bioinformatics. BMC Bioinformatics. 2008; 9(1):82. doi: 10.1186/1471-2105-9-82 PMID: 18251993

47. Cock PJA, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, et al. BioPython: Freely Available Python Tools for Computational Molecular Biology and Bioinformatics. Bioinformatics. 2009 Mar; 25 (11):1422–1423. doi: 10.1093/bioinformatics/btp163 PMID: 19304878

48. Hinsen K. The Molecular Modeling Toolkit: A New Approach to Molecular Simulations. Journal of Computational Chemistry. 2000 Jan; 21(2):79–85. doi: 10.1002/(SICI)1096-987X(20000130)21:2%3C79::AID-JCC1%3E3.0.CO;2-B

49. Hinsen K. High-Level Scientific Programming with Python. In: Proceedings of the International Conference on Computational Science-Part III. ICCS'02. London, UK, UK: Springer-Verlag; 2002. p. 691–700.

50. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms (3rd Edition). The MIT Press; 2009.

51. Jones NC, Pevzner PA. An Introduction to Bioinformatics Algorithms. The MIT Press; 2004.

52. Wünschiers R. Computational Biology: Unix/Linux, Data Processing and Programming. Springer-Verlag; 2004.

53. Model ML. Bioinformatics Programming Using Python: Practical Programming for Biological Data. O'Reilly Media; 2009.

54. Buffalo V. Bioinformatics Data Skills: Reproducible and Robust Research with Open Source Tools. O'Reilly Media; 2015.

55. Libeskind-Hadas R, Bush E. Computing for Biologists: Python Programming and Principles. Cambridge University Press; 2014.

56. Bassi S. A Primer on Python for Life Science Researchers. PLoS Comput Biol. 2007; 3(11):e199. doi: 10.1371/journal.pcbi.0030199 PMID: 18052533

57. Dudley JT, Butte AJ. A Quick Guide for Developing Effective Bioinformatics Programming Skills. PLoS Comput Biol. 2009 Dec; 5(12):e1000589. doi: 10.1371/journal.pcbi.1000589 PMID: 20041221

58. Noble WS. A Quick Guide to Organizing Computational Biology Projects. PLoS Comput Biol. 2009 Jul; 5(7):e1000424. doi: 10.1371/journal.pcbi.1000424 PMID: 19649301

59. Software Carpentry;. Accessed 2016-01-18. Available from: http://software-carpentry.org/.

60. Expressions—Python 3.5.1 documentation; 2016. Accessed 2016-01-18. Available from: https://docs.python.org/3/reference/expressions.html#operator-precedence.

61. Pierce BC. Types and Programming Languages. The MIT Press; 2002.

62. Hunter JD. Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering. 2007; 9 (3):90–95. doi: 10.1109/MCSE.2007.55

63. More Control Flow Tools—Python 3.5.1 documentation; 2016. Accessed 2016-01-18. Available from: https://docs.python.org/3.5/tutorial/controlflow.html#keyword-arguments.

64. McConnell S. Code Complete: A Practical Handbook of Software Construction (2nd Edition). Pearson Education; 2004.

65. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Pearson Education; 1994.

66. Zelle J. Python Programming: An Introduction to Computer Science. 2nd ed. Franklin, Beedle & Associates Inc.; 2010.

67. Mirabet V, Besnard F, Vernoux T, Boudaoud A. Noise and Robustness in Phyllotaxis. PLoS Comput Biol. 2012 Feb; 8(2):e1002389. doi: 10.1371/journal.pcbi.1002389 PMID: 22359496

68. van der Walt S, Schönberger JL, Nunez-Iglesias J, Boulogne F, Warner JD, Yager N, et al. scikit-image: image processing in Python. PeerJ. 2014; 2:e453. doi: 10.7717/peerj.453 PMID: 25024921

69. National Research Council (US) Committee on Frontiers at the Interface of Computing and Biology. On the Nature of Biological Data. In: Lin HS, Wooley JC, editors. Catalyzing Inquiry at the Interface of Computing and Biology. Washington, DC: The National Academies Press; 2005. Available from: http://www.ncbi.nlm.nih.gov/books/NBK25464.

70. Agirre E, Eyras E. Databases and Resources for Human Small Non-coding RNAs. Human Genomics. 2011 Mar; 5(3):192–199. doi: 10.1186/1479-7364-5-3-192 PMID: 21504869

71. De Keersmaecker SCJ, Thijs IMV, Vanderleyden J, Marchal K. Integration of Omics Data: How Well Does it Work for Bacteria? Molecular Microbiology. 2006 Dec; 62(5):1239–1250. doi: 10.1111/j.1365-2958.2006.05453.x PMID: 17040488

72. Phan JH, Quo CF, Cheng C, Wang MD. Multiscale Integration of -Omic, Imaging, and Clinical Data in Biomedical Informatics. IEEE Reviews in Biomedical Engineering. 2012; 5:74–87. doi: 10.1109/RBME.2012.2212427 PMID: 23231990

73. Wikipedia. Tree (data structure); 2016. Accessed 2016-01-18. Available from: https://en.wikipedia.org/wiki/Tree_%28data_structure%29.

74. Scipy. scipy.spatial.KDTree—SciPy v0.14.0 Reference Guide; 2014. Accessed 2016-01-18. Available from: http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html.

75. Wikipedia. k-d tree; 2016. Accessed 2016-01-18. Available from: https://en.wikipedia.org/wiki/K-d_tree.

76. Wikipedia. Graph (abstract data type); 2015. Accessed 2016-01-18. Available from: https://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29.

77. Hagberg AA, Schult DA, Swart PJ. Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008). Pasadena, CA USA; 2008. p. 11–15.

78. PyBrain. Journal of Machine Learning Research. 2010; 11:743–746.

79. Bloom BH. Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun ACM. 1970 Jul; 13(7):422–426. Available from: http://doi.acm.org/10.1145/362686.362692. doi: 10.1145/362686.362692

80. Moitzi M. bintrees 2.0.2; 2016. Accessed 2016-01-18. Available from: https://pypi.python.org/pypi/bintrees/2.0.2.

81. Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-Tree Filesystem. Trans Storage. 2013 Aug; 9(3):9:1–9:32. Available from: http://doi.acm.org/10.1145/2501620.2501623. doi: 10.1145/2501620.2501623

82. Wirth N. Algorithms + Data Structures = Programs. Prentice-Hall Series in Automatic Computation. Prentice Hall; 1976.

83. Budd T. An Introduction to Object-Oriented Programming. 3rd ed. Pearson; 2001.

84. Phillips D. Python 3 Object Oriented Programming. Packt Publishing; 2010.

85. Cieślik M, Derewenda ZS, Mura C. Abstractions, Algorithms and Data Structures for Structural Bioinformatics in PyCogent. Journal of Applied Crystallography. 2011 Feb; 44(2):424–428. doi: 10.1107/S0021889811004481 PMID: 22479120

86. The Self Variable in Python Explained;. Available from: http://pythontips.com/2013/08/07/the-self-variable-in-python-explained.

87. Why Explicit Self Has to Stay;. Available from: http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html.

88. Weininger D. SMILES: A Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules. Journal of Chemical Information and Modeling. 1988 Feb; 28(1):31–36. doi: 10.1021/ci00057a005

89. Pérez F, Granger BE. IPython: a System for Interactive Scientific Computing. Computing in Science and Engineering. 2007 May; 9(3):21–29. Available from: http://ipython.org. doi: 10.1109/MCSE.2007.53

90. Python Data Analysis Library;. Available from: http://pandas.pydata.org/.
91. Friedl JEF. Mastering Regular Expressions. O'Reilly Media; 2006.
92. Regexes on Stack Overflow;. Available from: http://stackoverflow.com/tags/regex/info.
93. Regex Tutorials, Examples and Reference;. Available from: http://www.regular-expressions.info.
94. Lee DY, McMurray CT. Trinucleotide Expansion in Disease: Why Is There a Length Threshold? Current Opinion in Genetics & Development. 2014 Jun; 26:131–140. doi: 10.1016/j.gde.2014.07.003
95. Brunger AT, Adams PD. Molecular Dynamics Applied to X-ray Structure Refinement. Accounts of Chemical Research. 2002 Jun; 35(6):404–412. doi: 10.1021/ar010034r PMID: 12069625
96. Langtangen HP. A Primer on Scientific Programming with Python. Texts in Computational Science and Engineering. Springer; 2014.
97. Jones E, Oliphant T, Peterson P, et al. SciPy: Open-source Scientific Tools for Python; 2001-. [Online; accessed 2015-06-30]. Available from: http://www.scipy.org/.
98. Scientific Computing Tools for Python;. Available from: http://www.scipy.org/about.html.
99. van der Walt S, Colbert SC, Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering. 2011 Mar; 13(2):22–30. doi: 10.1109/MCSE.2011.37
100. scikit-learn: machine learning in Python;. Available from: http://scikit-learn.org/.
101. Fortin FA, Rainville FMD, Gardner MA, Parizeau M, Gagné C. DEAP: Evolutionary Algorithms Made Easy. Journal of Machine Learning Research. 2012 jul;2171–2175(13:).
102. PyPI: The Python Package Index;. Available from: http://pypi.python.org.
103. Xia XQ, McClelland M, Wang Y. PypeR, A Python Package for Using R in Python. Journal of Statistical Software. 2010; 35(c02). Available from: http://EconPapers.repec.org/RePEc:jss:jstsof:35:c02.
104. rpy2, R in Python;. Available from: http://rpy.sourceforge.net.
105. Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The Best of Both Worlds. Computing in Science & Engineering. 2011 Mar; 13(2):31–39. doi: 10.1109/MCSE.2010.118
106. Cython: C-extensions for Python;. Available from: http://cython.org.
107. Open Source Initiative: Licenses & Standards;. Available from: http://opensource.org/licenses.
108. Morin A, Urban J, Adams PD, Foster I, Sali A, Baker D, et al. Shining Light into Black Boxes. Science. 2012 Apr; 336(6078):159–160. doi: 10.1126/science.1218263 PMID: 22499926
109. Blischak JD, Davenport ER, Wilson G. A Quick Introduction to Version Control with Git and GitHub. PLoS Comput Biol. 2016 Jan; 12(1):e1004668. doi: 10.1371/journal.pcbi.1004668 PMID: 26785377