

Introduction to Python: Basics

Mehmet Tevfik DORAK, MD PhD
Kingston University London
U.K.

Faculty of
Science,
Engineering
and Computing

Program

<i>Day 2</i>
<u>Introduction to Python as a coding language for bioinformatics</u>
What can you do with programming languages?
What is Python and why Python?
Python basics
Simple Python codes
Demonstrations of more advanced methods
Questions & answers

An Informal Introduction to Python

Python v3.1.5 documentation » The Python Tutorial »

[previous](#) | [next](#) | [modules](#) | [index](#)

Table Of Contents

- 3. An Informal Introduction to Python
 - 3.1. Using Python as a Calculator
 - 3.1.1. Numbers
 - 3.1.2. Strings
 - 3.1.3. About Unicode
 - 3.1.4. Lists
 - 3.2. First Steps Towards Programming

Previous topic

- 2. Using the Python Interpreter

Next topic

- 4. More Control Flow Tools

This Page

- [Report a Bug](#)
- [Show Source](#)

Quick search

Enter search terms or a module, class or function name.

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
SPAM = 1          # and this is the second comment
                  # ... and now a third!
STRING = "# This is not a comment."
```

3.1. Using Python as a Calculator

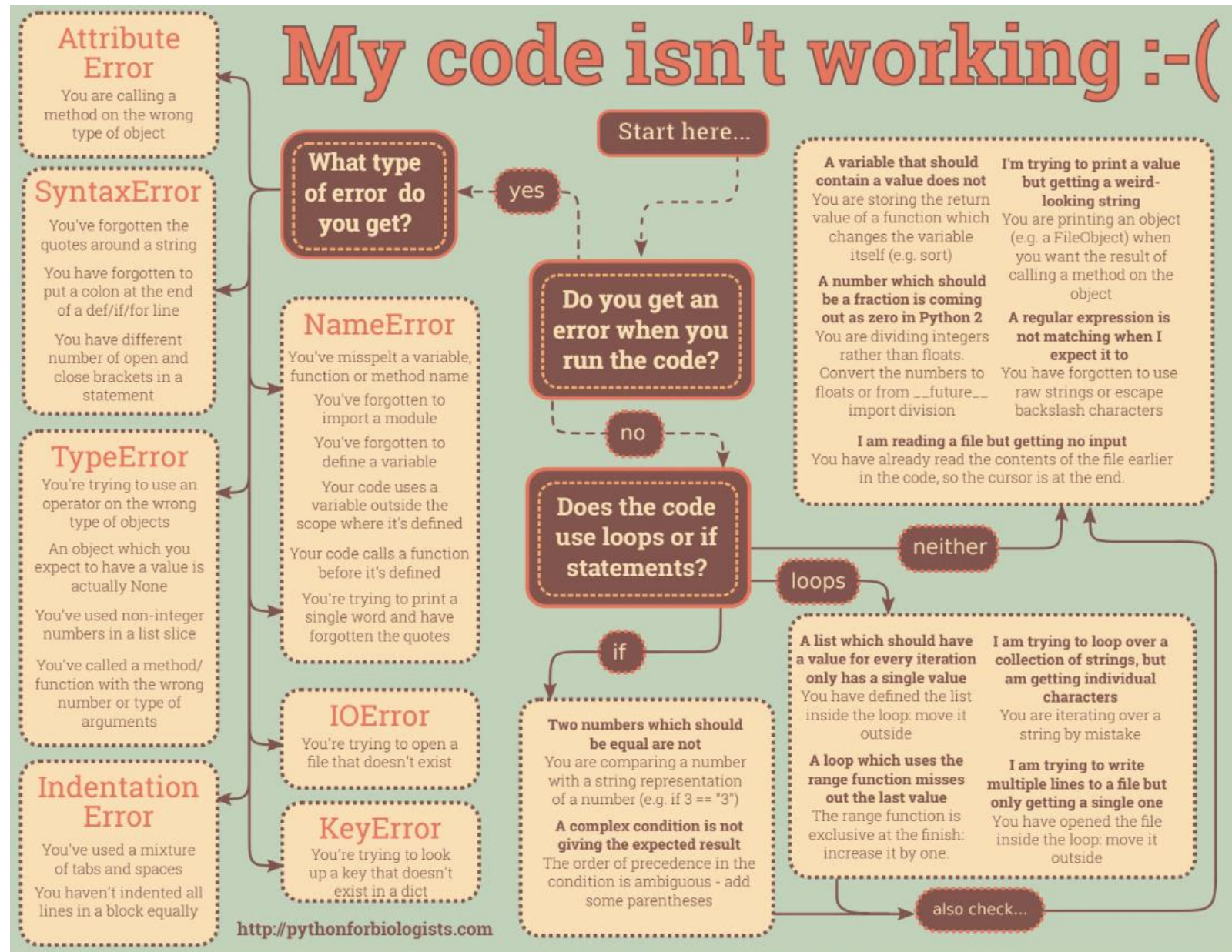
Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2 # and a comment on the same line as code
4
```

29 Common Beginner Python Errors



Python Data Types

NUMBERS

int, float, complex

SEQUENCES

string, list, tuple

SETS

MAPS

dictionaries

Sequence Data Types

Box 1. Most-Used Sequence Data Types

String: Usually enclosed by quotes (') or double quotes ("). Triple quotes (") are used to delimit multiline strings. Strings are immutable. Once created they can't be modified. String methods are available at <http://www.python.org/doc/2.5/lib/string-methods.html>.

For example:

```
>>> s0='A regular string'
```

List: Defined as an ordered collection of objects; a versatile and useful data type. C programmers will find lists similar to vectors. Lists are created by enclosing their comma-separated items in square brackets, and can contain different objects.

For example:

```
>>> MyList=[3,99,12,"one","five"]
```

This statement creates a list with five elements (three numbers and two strings) and binds it to the name "MyList". Each element of the list can be referred to by an integer index enclosed between square brackets. The index starts from 0, therefore MyList[3] returns "one". All list operations are available at <http://www.python.org/doc/2.5/lib/typesseq-mutable.html>.

Tuple: Also an ordered collection of objects, but tuples, unlike lists, are immutable. They share most methods with lists, but only those that don't change the elements inside the tuple. Attempting to change a tuple raises an exception. Tuples are created by enclosing their comma-separated items between parentheses. Tuples are similar to Pascal records or C structs; they are small collections of related data that are operated on as a group. They are used mostly for encapsulating function arguments, or any data that are tightly coupled.

For example:

```
>>> MyTuple=(2,3,10)
```

Tuple operations are available at <http://www.python.org/doc/2.5/lib/typesseq.html>.

OPEN ACCESS Freely available online

PLoS COMPUTATIONAL BIOLOGY

Education

A Primer on Python for Life Science Researchers

Sebastian Bassi

Sequence Data Types

Box 2. Unordered Types

Set: An unordered collection of immutable values. It is mostly used for membership testing and removing duplicates from a sequence. Sets are created by passing any sequential object to the set constructor, such as: `set([1,2,3])`

For more information on sets, please refer to <http://www.python.org/doc/2.5/lib/types-set.html>.

For example:

```
>>> ResEzSet1=set(['BamH1', 'HindIII', 'EcoR1', 'Sall'])
>>> ResEzSet2=set(['PlaA', 'EcoR1', 'Eco143'])
>>> ResEzSet1&ResEzSet2
set(['EcoR1'])
```

Dictionary:

A data type that stores unordered one-to-one relationships between keys and values. Unordered in this context means that each key-value pair is stored without any particular order in the dictionary. It is analogous to a hash in Perl or a Hashtable class in Java. Dictionaries are created by placing a comma-separated list of key-value pairs within braces.

For example:

Set Translate as a dictionary with codon triplets as keys and the corresponding amino acids as values:

```
>>> Translate={"cca":"P","cag":"Q","agg":"R"}
```

Creating a new entry:

```
>>> Translate["gat"]="D"
```

To see what is inside the dictionary:

```
>>> Translate
{'agg': 'R', 'cag': 'Q', 'gat': 'D', 'cca': 'P'}
```

Dictionaries share some methods with lists. A complete list of methods on can be seen at: <http://www.python.org/doc/2.5/lib/typesmapping.html>.

Python Arithmetic Operators

Addition: +

Subtraction: -

Multiplication: *

Exponent: **

Division: /

Floor division: //

Modulus: %
(returns remainder)

Numeric Data Types

Table 1. Numeric Data Types

Numeric Data Type	Description	Example
Integer	Holds integer numbers without any limit (besides your hardware). Some texts still refers to a “long integer” type since there were differences in usage for numbers higher than $2^{32} - 1$. These differences are now reserved for language internal use.	42, 0, -77
Float	Handles the floating point arithmetic as defined in ANSI/IEE Standard 754 [43], known as “double precision.” Internal representation of float numbers is not exact, but accurate enough for most applications. The “decimal” module [44] has more precision at the expense of computing time.	424.323334, 0.00000009, 3.4e-49
Complex	Complex numbers are the sum of a real and an imaginary number and is represented with a “j” next to the real part. This real part can be an integer or a float number. j is the notation for imaginary number, equal to the square root of -1.	$3 + 12j$, j 9j
Boolean	True and False are defined as values of the Boolean type. Empty objects, None, and numeric zero are considered False. Objects are considered True.	False, True

doi:1011371/journal.pcbi.0030199.t001

Control Flow

Box 3. Control Structures

If statement: Tests for a condition and acts upon the result of that condition. If the condition is true, the block of code after the “if condition” will be executed. If it is false, the program will skip that block and will test for the next condition (if any). Several conditions can be tested using **elif**. If all conditions are false, the block under **else** will be executed. **Elif** can be used to emulate a C “switch-case” statement. Scheme of an if statement:

```
if condition1:
    block1
elif condition2:
    block2
else:
    block3
```

For loop: Iterates over all the members of a sequence of values (as in Perl’s “foreach”). It is different from C and VB because there isn’t a variable that increments or decrements on each cycle. This sequence could be any type of iterable object like a list, string, tuple, or dictionary. The code inside a for loop will be executed once for each item in the sequence, and at the same time the variable will take the value of each item in the sequence. There could be an optional **else** clause. If it is present, the block under the else clause is executed when the loop terminates through exhaustion of the list, but not when the loop is terminated by a **break** statement.

The structure of a for loop is:

```
for variable in sequence:
    block1
else:
    block2
```

while loop: Executes a block of code as long as a condition is true. As the for loop, there could be an optional **else** clause. When present, the block under the else clause is executed when the condition becomes false but not when the loop is terminated by a **break** statement.

The general form is:

```
while condition:
    block1
else:
    block2
```

OPEN ACCESS Freely available online

PLoS COMPUTATIONAL BIOLOGY

Education

A Primer on Python for Life Science Researchers

Sebastian Bassi

Python for Bioinformatics



oreilly.com Safari Books Online Conferences

Search

advertisement

Content

All Articles
Python News
Numerically Python
Python & XML

Python Topics

Community
Database
Distributed
Education
Getting Started
Graphics
Internet
OS
Programming
Scientific
Tools
Tutorials
User Interfaces

ONLamp Subjects

Linux
Apache
MySQL
Perl
PHP

Beginning Python for Bioinformatics

by [Patrick O'Brien](#)
10/17/2002

Print

[Subscribe to Python](#)

[Subscribe to Newsletters](#)

Bioinformatics, the use of computers in biological research, is the newest wrinkle on one of the oldest pursuits—trying to uncover the secret of life. While we may not know all of life's secrets, at the very least computers are helping us understand many of the biological processes that take place inside of living things. In fact, the use of computers in biological research has risen to such a degree that computer programming has now become an important and almost essential skill for today's biologists.

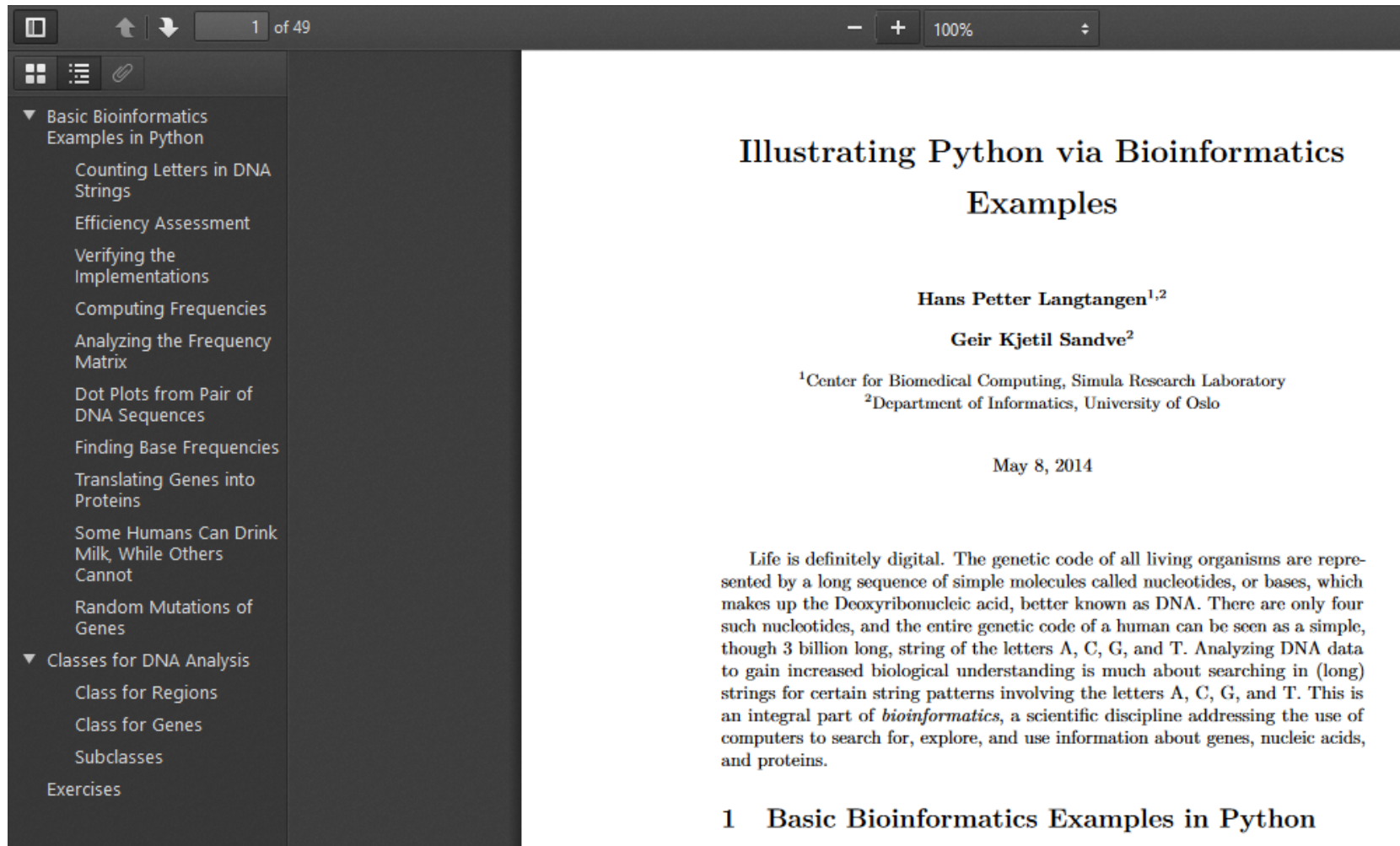
The purpose of this article is to introduce Python as a useful and viable development language for the computer programming needs of the bioinformatics community. In this introduction, we'll identify some of the advantages of using Python for bioinformatics. Then we'll create and demonstrate examples of working code to get you started. In subsequent articles we'll explore some significant bioinformatics projects that make use of Python.

A Bit of Background

Because scientists have long relied on the open availability of each other's research results, it was only natural that they would turn to Open Source software when it came time to apply computer processes to the study of biological processes. One of the first Open Source languages to gain popularity among biologists was Perl. Perl gained a foothold in bioinformatics based on its strong text processing facilities, which were ideally suited to analyzing early sequence data. To its credit, Perl has a history of successful use in bioinformatics and is still a very useful tool for biological research.

In comparison to Perl, Python is a relative newcomer to bioinformatics, but is steadily gaining in popularity. A few of the reasons for this popularity are the:

Python for Bioinformatics



The image shows a presentation slide with a dark-themed sidebar on the left and a main content area on the right. The sidebar contains a list of topics under 'Basic Bioinformatics Examples in Python' and 'Classes for DNA Analysis'. The main content area displays the title 'Illustrating Python via Bioinformatics Examples', the authors 'Hans Petter Langtangen^{1,2}' and 'Geir Kjetil Sandve²', their affiliations, the date 'May 8, 2014', and a paragraph of text about the genetic code. The slide is numbered '1 of 49' in the top left corner of the presentation window.

1 of 49

Illustrating Python via Bioinformatics Examples

Hans Petter Langtangen^{1,2}
Geir Kjetil Sandve²

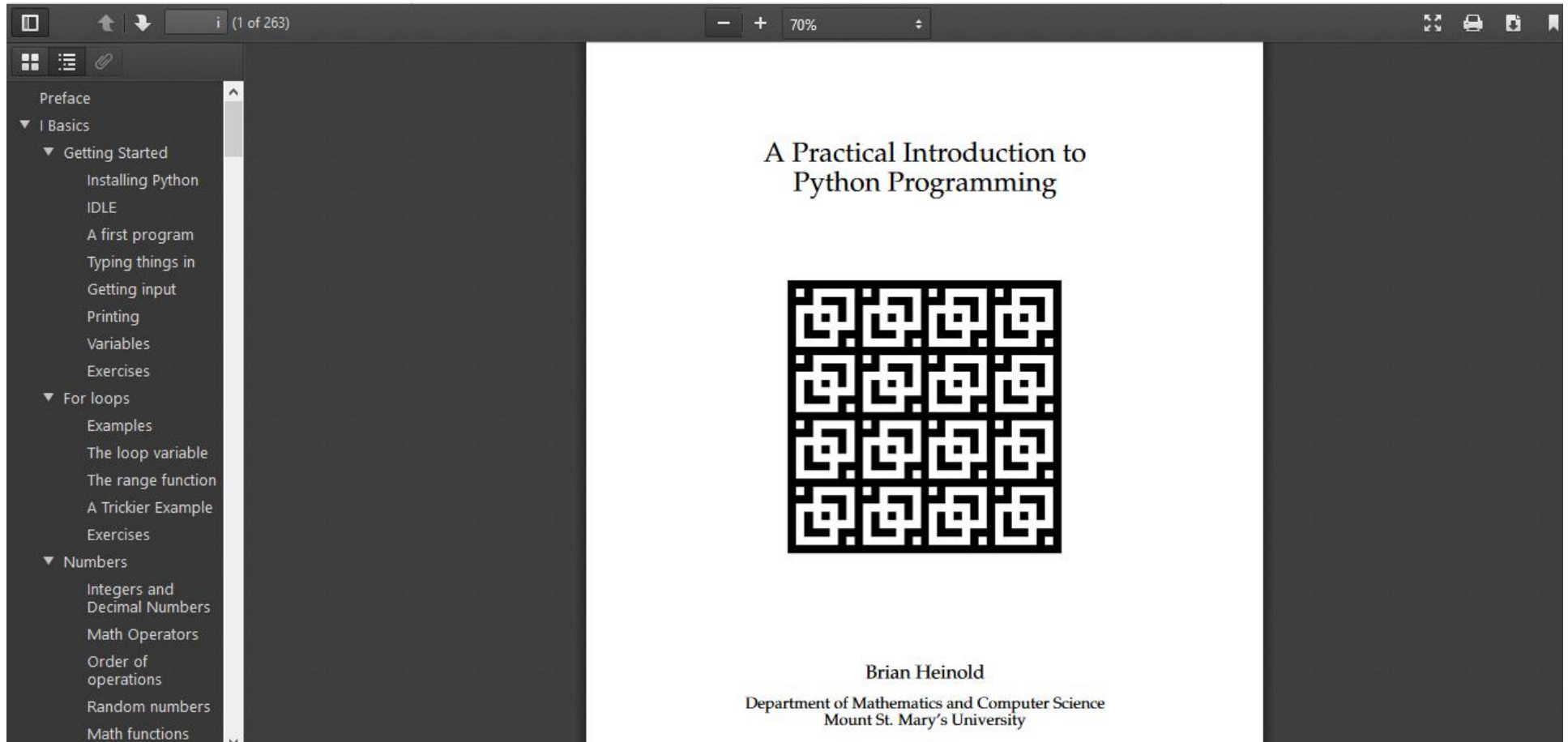
¹Center for Biomedical Computing, Simula Research Laboratory
²Department of Informatics, University of Oslo

May 8, 2014

Life is definitely digital. The genetic code of all living organisms are represented by a long sequence of simple molecules called nucleotides, or bases, which makes up the Deoxyribonucleic acid, better known as DNA. There are only four such nucleotides, and the entire genetic code of a human can be seen as a simple, though 3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain increased biological understanding is much about searching in (long) strings for certain string patterns involving the letters A, C, G, and T. This is an integral part of *bioinformatics*, a scientific discipline addressing the use of computers to search for, explore, and use information about genes, nucleic acids, and proteins.

1 Basic Bioinformatics Examples in Python

Python for Bioinformatics



Python for Biologists: Tutorial



HOME PYTHON BOOKS TRAINING COURSES PROGRAMMING ARTICLES PYTHON TUTORIAL

Python tutorial

INTRODUCTION

PRINTING AND
MANIPULATING TEXT

WORKING WITH FILES

LISTS AND LOOPS

WRITING OUR OWN
FUNCTIONS

CONDITIONAL TESTS

REGULAR
EXPRESSIONS

DICTIONARIES

FILES, PROGRAMS,
AND USER INPUT

WHY PYTHON?

The importance of programming languages is often overstated. What I mean by that is that people who are new to programming tend to worry far too much about what language to learn. The choice of programming language does matter, of course, but it matters far less than most people think it does. To put it another way, choosing the "wrong" programming language is very unlikely to mean the difference between failure and success when learning. Other factors (motivation, having time to devote to learning, helpful colleagues) are far more important, yet receive less attention.

The reason that people place so much weight on the "what language should I learn?" question is that it's a big, obvious question, and it's not difficult to find people who will give you strong opinions on the subject. It's also the first big question that beginners have to answer once they've decided to learn programming, so it assumes a great deal of importance in their minds.

There are three main reasons why choice of programming language is not as important as most people think it is. Firstly, nearly everybody who spends any significant amount of time programming as part of their job will eventually end up using multiple languages. Partly this is just down to the simple constraints of various languages – if you want to write a web application you'll probably do it in Javascript, if you want to write a graphical user interface you'll probably use something like Java, and if you want to write low-level algorithms you'll probably use C.

Secondly, learning a first programming language gets you 90% of the way towards learning a second, third, and fourth one. Learning to think like a programmer in the way that you break down complex tasks into simple ones is a skill that cuts across all languages – so if you spend a few months learning Python and then discover that you really need to write in C, your time won't have been wasted as you'll be able to pick it up much quicker.

Thirdly, the kinds of problems that we want to solve in biology are generally amenable to being solved in any language, even though different programming languages are good at different things. In other words, as a beginner, your choice of language is vanishingly unlikely to prevent you from solving the problems that you need to solve.

<https://pythonforbiologists.com>

... Looking forward

<i>Day 2</i>
<u>Introduction to Python as a coding language for bioinformatics</u>
What can you do with programming languages?
What is Python and why Python?
Python basics
Simple Python codes
Demonstrations of more advanced methods
Questions & answers

