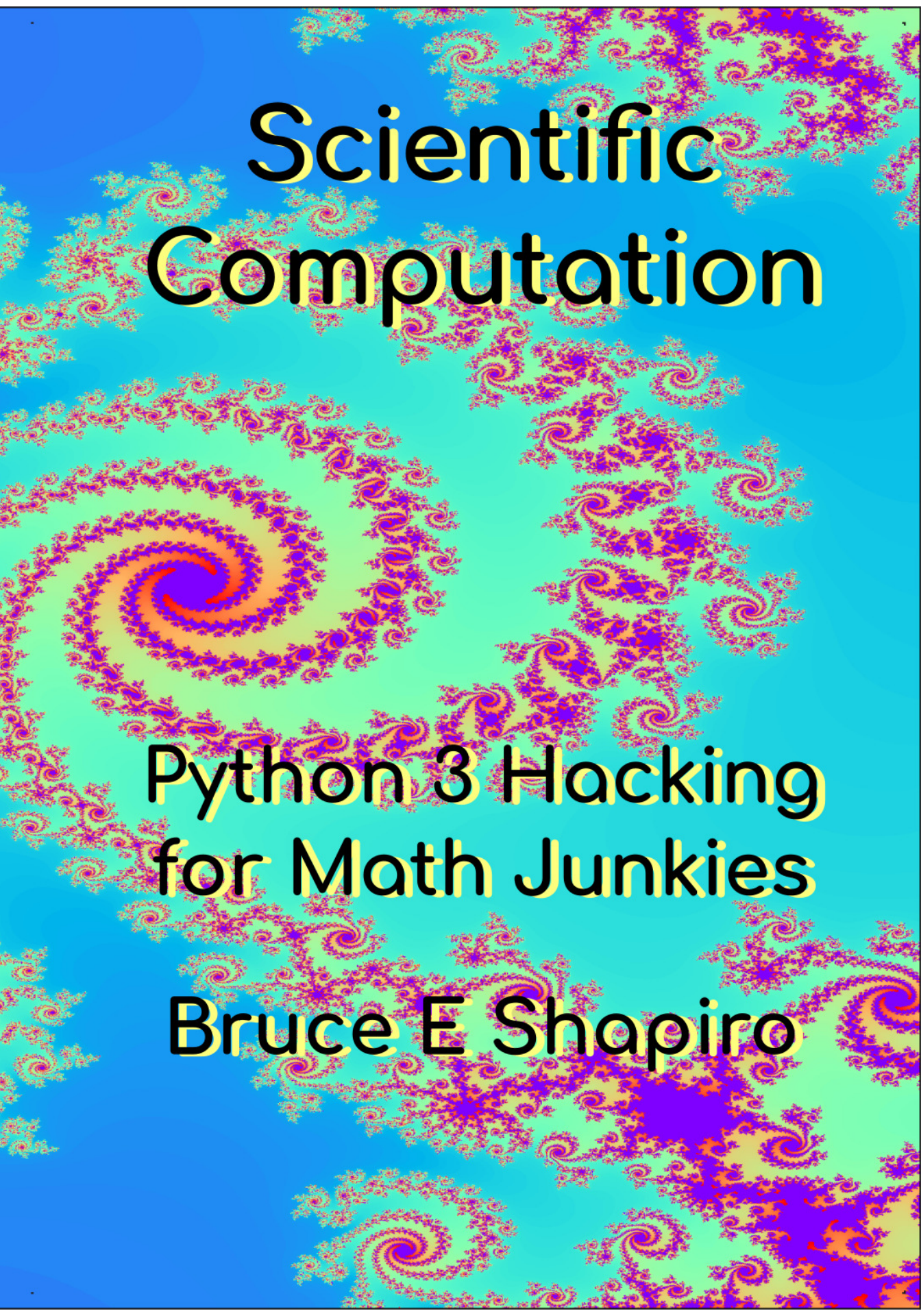

Scientific Computation
Python 3 Hacking for Math Junkies
Bruce E Shapiro

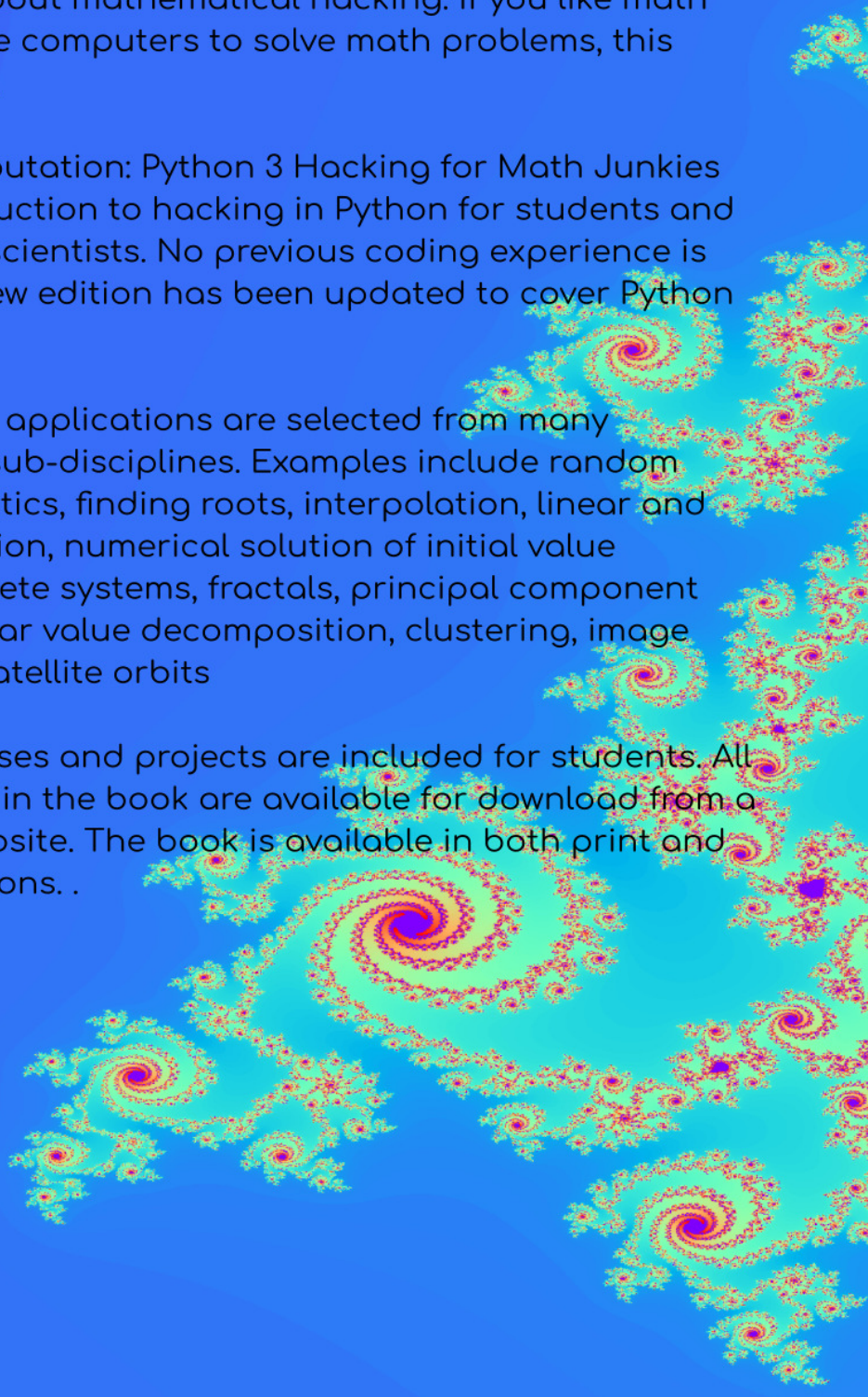
Scientific Computation: Python 3 Hacking for Math Junkies

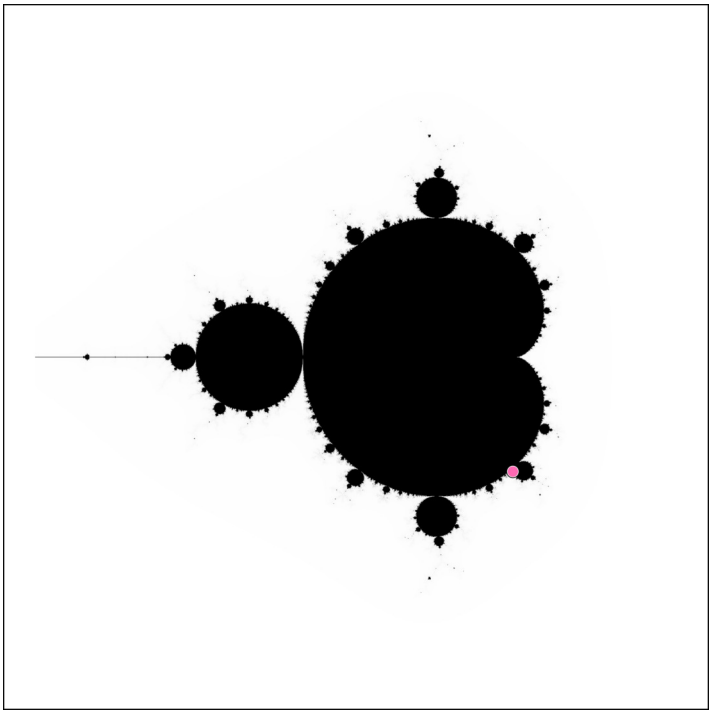
This is a book about hacking, but not just any kind of hacking. It is about mathematical hacking. If you like math and want to use computers to solve math problems, this book is for you.

Scientific Computation: Python 3 Hacking for Math Junkies gives an introduction to hacking in Python for students and mathematical scientists. No previous coding experience is needed. This new edition has been updated to cover Python version 3.

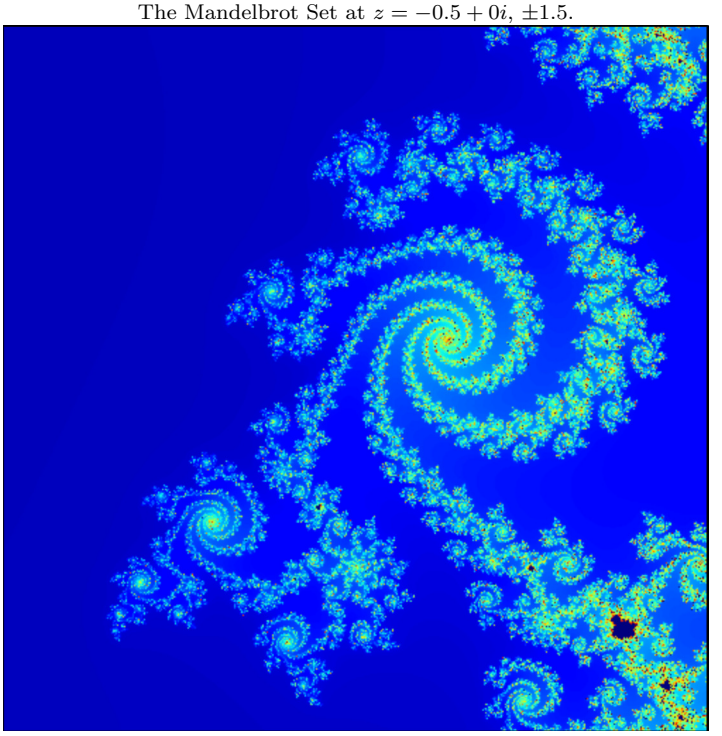
Computational applications are selected from many mathematical sub-disciplines. Examples include random numbers, statistics, finding roots, interpolation, linear and logistic regression, numerical solution of initial value problems, discrete systems, fractals, principal component analysis, singular value decomposition, clustering, image analysis, and satellite orbits

Over 300 exercises and projects are included for students. All code examples in the book are available for download from a companion website. The book is available in both print and electronic versions. .





The Mandelbrot Set at $z = -0.5 + 0i$, ± 1.5.



The Mandelbrot Set at $z = 0.2323 - 0.5345i$, ± 0.0025

The Boundary Between Light and Dark

The interior of the Mandelbrot set consists of all those points c in the complex plane for which fixed point iteration on the function $f(z) = z^2 + c$ converges. This region is filled with black in the figure on the top of the previous page.

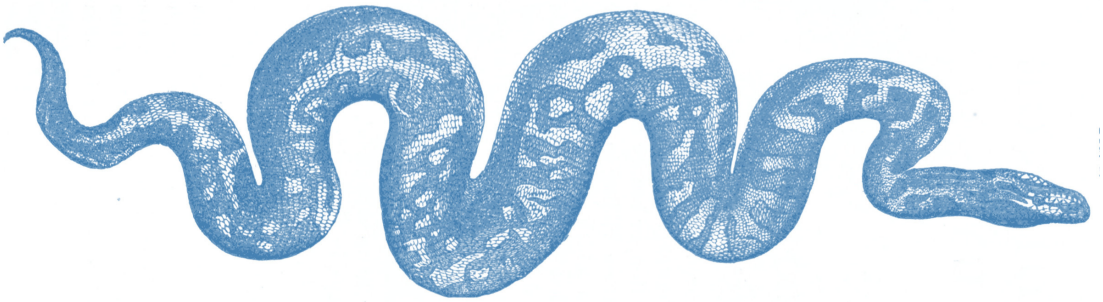
The Mandelbrot set displays the properties of fractals: it is bounded; its edge has infinite self similarity; and its boundary has an infinite length.

A wide variety of interesting phenomenon occur on the boundary between light and dark, the edge of the Mandelbrot Set. The transition is dramatic in black and white, but seems almost transcendent when we map the number of iterations to a color scale. Mapping the region around the point $z = 0.2323 - 0.5345i$ (location indicated by a red dot on the top image on the previous page) in this way produces the front cover image.

About the Author

Bruce E. Shapiro teaches mathematics at California State University, Northridge, in the most challenging job he has ever had. For a while he collected degrees from various colleges, until his wife said uncle. In past lives he has been called a rocket scientist, a brain scientist, a computational scientist, a mathematical scientist, a data scientist, and a generally annoying and snarky pain in the ass. None of those pursuits were particularly challenging. Now he spends his spare time eating chocolate chip cookies, playing with imaginary friends in the complex plane, and pontificating about the poor state of public education in California while teaching his two Staffordshire Terriers, Bella and Romeo (pictured above), the finer points of machine learning.

Scientific Computation



Python 3 Hacking for Math Junkies

With Jupyter Notebooks

Bruce E. Shapiro

Fourth Edition

2018

Sherwood Forest

http://CalculusCastle.com

Scientific Computation: Python 3 Hacking for Math Junkies
Fourth Edition, Version 4.0.427 (revised 8/19/18)
Bruce E. Shapiro, Ph.D., Sherwood Forest Books, Los Angeles, CA, USA

Earlier versions of this book were published under the title *Scientific Computation: Python Hacking for Math Junkies*.

ISBN-13 978-1725894662
ISBN-10 1725894661

The drawing of the Python is from "Concerning Serpents" by Elias Lewis, Jr, *Popular Science Monthly* **4**(17):258-275 (Jan. 1874), published in the United States of America by D. Appleton & Co. The copyright has expired and this image is in the public domain.

The picture of the author in *About the Author* was drawn by A. Babahekian.

The cover image of the Mandelbrot Set was generated with the Python code on the back cover.

To make a suggestion visit https://github.com/biomathman/python-book/issues. Please include the version number and revision date listed at the top of this page on all error reports. To see a list of known errata visit https://github.com/biomathman/python-book/wiki. Color versions of the figures and jupyter notebooks with all the examples in the book may be obtained from the github repository.

Many of the software packages described in this book are copyright by their respective development teams. Please visit the web pages referenced within the text to determine the appropriate copyright and distribution permissions for each package.

Many grateful thanks are extended to readers for the useful suggestions they have provided and errors and omissions they have caught and and reported to me. ☺

Contents

The Hacker's Codes

First Generation

- Computer access shall be unrestricted.
- Information shall be free.
- Judge others **only** by their acts.
- Computers shall produce beauty.
- Computers shall improve life.

Second Generation

- Do no evil.
- Protect data.
- Protect privacy.
- Conserve resources.
- Telecommunication shall be unrestricted.
- Share code.
- Strive to improve.
- Be prepared for cyber-attack.
- Always improve security.
- Always test and improve the system.

List of Tables

Configuration

Basic Operations

Comparisons

Sequences

Lists

Strings

Sets

Numpy Arrays

Plotting with Pyplot

Dictionaries

Enumeration

Python Classes

Probability

Statistics

Linear Algebra

Spatial Geometry

Interpolation

Differential Equations

Image Processing

Maps and Geography

Preface

Hacking

Exploring the limits of what is possible, in a spirit of playful cleverness.[a]

[a] Richard Stallman. "On Hacking." http://stallman.org/articles/on-hacking.html

This is a book about hacking, but not just any kind of hacking. It is about *mathematical hacking*, or *scientific computing*. If you like math and want to use computers to do math or solve mathematical problems, then this book is for you.

Why Python?

Most importantly, it's free.[1] You can install it on any computer.

Python is standardized.[2] The programs you write on one computer can be run on any other computer, regardless of operating system.

Python is generic. The tools and techniques that it utilizes are similar to ones found in many major programming languages. Once you learn Python, you can learn any language.

There are a lot, and I mean a lot, of mathematical and scientific libraries available. The hard stuff has most likely already been coded for you.[3]

Python is multipurpose: you can write applications, run command shell programs, or run from within a closed environment. Python can be used as a scripting program and easily run programs or access libraries written in other languages. It has access to a wide variety of graphical user interfaces.[4]

Python notebooks allow you to integrate full programs, text, markup, input, output, and graphics into single files that can be easily made visible on the web.

Basically, Python provides all things to all people, ranging from an easy interface for the newbie to advanced development interfaces for programmers.

Oh, did I mention that Python is free?

[1] In every sense of the word, as in both *free speech* and *free beer*.
[2] By the Python Software Foundation. See https://www.python.org/.
[3] Check out http://central.scipy.org/ and https://www.scipy.org/topical-software.html to get an idea.
[4] Experienced programmers may want to check out https://wiki.python.org/moin/GuiProgramming.

Scientific Computing is not Exactly Programming

Computer programming is largely about design: readability and reusability. These things are important if you are developing a commercial product.[5] Functionality is often only a small consideration in the software development process.

Hacking, on the other hand, is all about functionality and elegance.[6] It is not very different from proving theorems.

A good proof is the most elegant, logical sequence of steps that takes you from assumption to conclusion.

A good hack is the most elegant, concise sequence of steps that takes you from input to output.

In a proof, the steps follow logically, and follow the rules of logic.

If a hack, the steps must be semantically correct, and have the correct syntax.

Many scientific programs are written to be run only once and never again. Scientific programs often only have a single customer. The customer and programmer are frequently the same person.

Niceties like idiot-proofing and fancy interfaces, which may consume the vast majority of the code in a commercial program, become unnecessary when you are your own customer. Even if you are writing a program for a large scientific lab, and that program is going to be used repeatedly by many people, such things may not be considered so important.[7] Documentation becomes replaced by in-line comments and web pages. Copies of the source code go into either private local or public open access repositories for the community or world to see, redact, and improve.

Why *this* book?

There are sooooooooooooo many books on Python.[8] There are even more books on numerical analysis and scientific computing.[9] There is not much overlap.[10] The computing books generally assume you already know programming, and the programming books have very little to give to mathematical scientists. Into this bifurcation comes this *Math Hacker's Guide to the Pythonverse*,[11] which should also help pay for my retirement, so I can continue to live on a healthy diet of spam and eggs.

[5] When your company is sold and the new owners dump the entire development staff so that they can outsource software support to a lower bidder, you have nobody to blame except for yourself for writing such clear and clean code. See, for example, http://www.theregister.co.uk/2016/01/27/vmware_fusion_and_workstation_development_team_fired/.

[6] To learn more about hacking and the Hacker's codes, see Levy S. (2010) *Hackers*. Sebastopol, CA: O'Reilly Media; Mizrach, S. "Is there a hacker ethic for 90s Hackers?" http://www2.fiu.edu/~mizrachs/hackethic.html

[7] If you want to sell an app commercially, these are extremely important, of course. Nobody is going to *buy* something they can't understand or which crashes unexpectedly. But if you are the only one using it, you know the pitfalls of the program and how to avoid them.

[8] 2486 listings were returned by Amazon for books on the "python computer language" (27 June 2016).

[9] 10,747 listings for "scientific computing," 42,660 for "numerical analysis."

[10] There were only 20 books were listed under "numerical analysis python" and 193 books under "scientific computing python," including earlier editions of this book.

[11] With apologies to the late Douglas Adams. I thought of using this as a book title, but the mixed metaphor with Monty Python didn't really work.

I have included examples from a variety of disciplines that are usually relegated to advanced treatises on numerical analysis, machine learning, and the applied sciences, and have kept the amount of proof to a minimum. Some additional topics are introduced in the exercises. Nearly all of the algorithms presented have come from my professional research experience.

Much of the material on Python is summarized in over eighty tables (listed on pages v and vi). It would not have been possible to cover all of this material in very much detail in a single text, and yet understanding the possibilities and the extent of this content is precisely what students need. Even so, it remains far from comprehensive: many important packages, such as date, time, and operating system interfaces, had to be omitted. Thus users are referred to the official documentation for further details. The goal of the tables is to collect some of the more relevant material. This will help target more directed online searches.

Why This Book Upgrade?

This book is an upgrade of my book *Scientific Computation: Python Hacking for Math Junkies* to cover Python 3. When I first started writing the earlier text, many of the scientific packages for Python either did not work or had not yet been ported to Python 3. Thus is made perfect sense to ignore Python 3 a few years ago.

This is no longer the case. These days, I run all my code in Python 3 and there is very little that does not work (at least without a little tweaking) in Python 3. All of the major packages work, and that is what is most important. Furthermore, Python 2.7, the terminal supported version of Python 2, is scheduled to be retired on 1 Jan 2020. This date is quickly approaching. While Python 2 will, in fact, be around and usable for many decades to come, the lack of official support means that while Python 3 will continue to grow (along with the number of libraries that users can access), the same will most likely not be true for Python 2.

I encourage all new Python students to start with Python 3, completely bypassing Python 2. Experienced hackers with a lot of Python 2 code will have a learning curve, as Python 3 is less forgiving. It is something less of a hacker's language, and more of a programmer's language. While I understand the modifications to the language, I find this somewhat sad, as some previously elegant code has to be re-written in slightly bulkier format.

About Jupyter Notebooks

There is a whole chapter about Jupyter Notebooks in this book, as before. Since this book is about hacking, and not about software development, I encourage students to do all of their work in these notebooks. There are two main advantages:

- In the classroom environment, this allows students to turn in an entire assignment with both input and output as a single file, that I can the run on my own machine.
- In the work world, students can build documents that contain work flows. Work flows include documentation, input, and output. These work flows can be converted to web pages for online publication.

Furthermore, there are many different kernels for Jupyter. Three of them were built into the name of the program: JUlia, PYThon, and R. I encourage students to learn all of these languages. This is because not all languages are appropriate for each task. I would not suggest writing a compiler or an embedded security system in R; nor would I suggest performing statistical analyses in Java, or doing web applications in C. The more languages a student has in their programming toolkit, the more ways they can solve a problem efficiently.

The Art of the Possible

"Politics is the art of the possible."[12] I've sometimes wondered if Stallman had that thought in mind when he defined hacking in his web article. Social media, for example, is changing the world in ways that Bismarck could not have even conceived.

More likely it was the voice of Juan Peron from *Evita* singing in his head:

> One has no rules, is not precise.
> One rarely acts the same way twice.
> One spurns no device, practicing the art of the possible.[13]

Either way, it makes you think.

Bella does Python.

[12] Die Politik ist die Lehre vom Möglichen (Otto von Bismarck, 1867).
[13] Andrew Lloyd Webber & Tim Rice (1979).

Notation

Typewriter Bold font is used to write Python code or command line expressions.

Full lines or multiple lines of code are enclosed in shaded boxes.

```
this is a line of code
```

```
>>> this is a line of code in the Python shell
```

```
$ this is an instruction typed into the Command shell
```

Input and output cells in the iPython shell are numbered. Even though cells in the iPython shell do not normally have outlines drawn around them, we have done so here. Print cells are also boxed, but not numbered.

```
In [1]: math.sqrt(4)
```

```
Out[1]: 2
```

Cells in iPython notebooks are similarly annotated

```
In [2]: x=math.sqrt(4)
        print(x, x**2)
        x**3
```

```
2 4
```

```
Out[2]: 8
```

Generic input cells, where variables do not have specific values, may not be given line numbers. They will be shown in unevaluated form.

```
In []: def f(x):
           # ... lines omitted - figure out y here
           return g(y)
```

Algorithms look like this, and may use statements like **repeat**, **while**, **for**, and **if**.

Algorithm 0.1 The name of the algorithm.

input: The input variables are listed here
1: $x \leftarrow a + b$
2: **repeat**
3: do stuff
4: **until** some condition is met
5: **return** some value

The underbracket character "␣", when used, represents a blank space inside of a string literal.

The forward arrow symbol $\rightarrow$ may be used either in algorithms or to explain how some functions or code snippets work. It means "this is the output of something." For example,

```
print(x+y) → 23
```

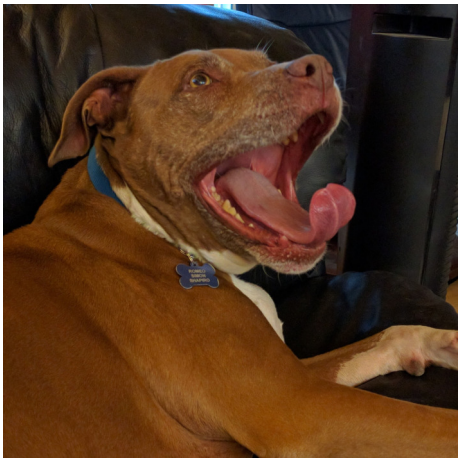
means: "if you type `print(x+y)` into the command shell now, then Python will return a value of `23`."

An assignment statement in an algorithm, such as

$$x \leftarrow a + b$$

means: "calculate the value of $a + b$ and then replace x with the result."

Within mathematical expressions, scalar variables and functions, such as x y, and $f(x, y)$ are typeset in a *Roman italic* font. Vectors, matrices, and points such as $\mathbf{v}$, $\mathbf{M}$, and $\mathbf{P}$ are shown in san serif bold. Standard mathematical notation is used throughout, e.g., curly braces $\{\}$ for sets, $\mathbb{R}$ for the real numbers, $\varnothing$ for the empty set, $\mathbf{M}^{-1}$ and $\mathbf{M}^{\mathrm{T}}$ for matrix inverse and transpose, etc.



Romeo cogitates on the art of the possible.

The Student's Imperative

Go forth and hack: but do no evil.

Part I

Getting Started

Part I provides a brief introduction to computing. No previous background in the subject is assumed.

Chapter 1 sets the stage, defining the basic terminology used later in the text. Everyone should read this.

Chapter 2 provides an overview of the Python language. This is followed in chapters 4, 5 and 6 with introductions to the "official" environments: the Python shell, the iPython shell, and iPython notebooks. Assuming that you are reading this book because you are new to Python, you should probably read all three of these chapters.

The Babylonian Algorithm for finding a root is discussed in chapter 3. This algorithm will be used as a canonical example throughout the textbook. If you are not already familiar with this technique, then you should read this chapter as well.

Chapters 7, 8 and 9 discuss some of the vocabulary that is commonly used in computing, involving numbers, errors, and convergence of algorithms. This material can be skipped o a first reading as it is not referenced extensively in the remainder of the text.

Chapter 1

Programs and Programming

English (or any other language, for that matter) has rules about putting words together to form sentences.[1] If the words are not in the right order, or the inflection (in spoken language) is imprecise, the meaning will be altered.

When you misspeak, people can usually figure out what you meant to say. For example, if you say "Please have napkin" to you friend during lunch, she will most likely pass you a napkin, figuring out that you meant to say "May I please have a napkin."

To tell a computer what to do you give it instructions in a **computer language**. The grammar and syntax of a computer language are precise, and do not leave any room for error. When you click on an application on you desktop, or an app on you cell phone, you are, in fact, **executing**[2] a computer program that has been written in some computer language. The author of the app has done the work of figuring out the necessary syntax for you; but by doing this, the programmer has traded the **flexibility** of full control over the computer for the **convenience** of mouse clicks.

The name of the language that we will use in this text is **Python**.[3] Python is just one of many different computer languages. There are so many different computer languages that the Wikipedia has a list of lists of computer languages.[4] When we write a computer language like Python, our sentences are called **statements** and our conversations are called **programs**. A **computer program** is nothing more than a collection of precisely formatted statements telling the computer what we want it to do.

Definition 1.1. Program

A **Computer Program** is a sequence of instructions for a computer, so that it can perform a specific task.

It is useful to visualize your computer as comprising several **layers of software** that overlay the physical **computer hardware** (figure 1.1). A program in any layer can be designed to access a program at any lower layer via **hooks**, special computer programming interfaces, written into the intermediate layers.

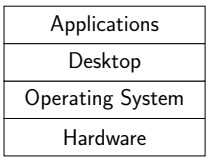
[1] If you have experience programming or understand the software development process, this chapter can be skipped without loss of continuity.

[2] When a computer program "does its thing" we say that that we are *executing* the program, derived from the verb *to execute*, to carry out or put into effect a course of action.

[3] All computer languages are given their names for *very important reasons*. For example, the early languages FORTAN and COBOL were short for "Formula Translator" and "Common Business Oriented Language," respectively. The creator of Python liked to watch *Monty Python's Flying Circus* on TV.

[4] As of 14 June 2016,there were twenty lists listed at https://en.wikipedia.org/wiki/Category:Lists_of_programming_languages. The alphabetical list of languages at https://en.wikipedia.org/wiki/List_of_programming_languages showed 691 but excluded over 300 dialects of BASIC and esoteric languages.

Figure 1.1: Several layers of computer software may be envisioned to sit on top of the physical computer hardware, with the operating system closest to the hardware and the application software furthest away.

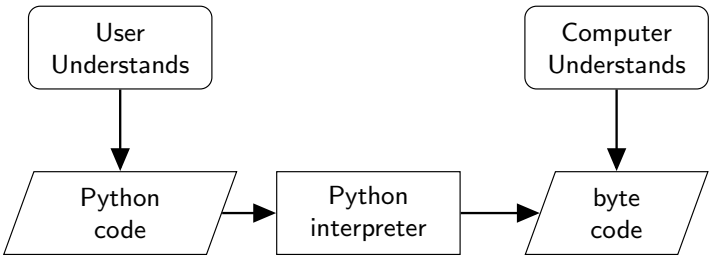


Computer programs can be broadly grouped into two categories:

- **Application programs**, programs designed for the "end user, "[5] such as spreadsheets, word processors, presentation programs, and web browsers.
- **System software**, which encompasses anything that makes the computer work, including the **operating system** and all the hardware interfaces. Most system software is invisible and runs in the background.

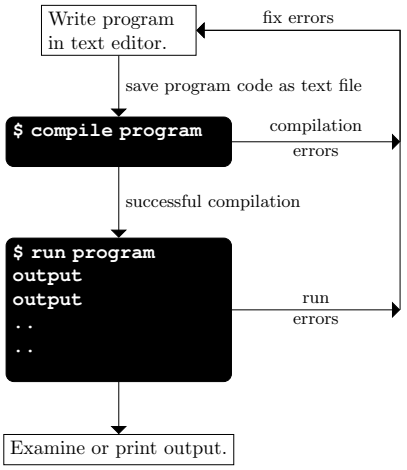
An exception is the **desktop environment**. Desktop environments control the look and feel of user-computer interface, through things like menus, icons, folders, task bars and launchers. Most Windows and Mac users identify their operating systems by their desktops, e.g., the Mac Desktop or the Windows Desktop. However, the desktop is only a small, though highly visible, part of the operating system, In most cases it is possible to replace the desktop environment on any computer while leaving the remainder of the operating system essentially intact. Linux users have a variety of desktop environments to choose from, with names like gnome, KDE, x-windows, and so forth; many even choose to install multiple desktops on their computer and switch back and forth between them. In theory you could configure your Windows desktop to look like Ubuntu.[6]

Figure 1.2: When you invoke the Python interpreter, it first looks to see if a byte code version has already been created. The byte code versions is all ones and zeros and you cannot read it. If it does not exist, it will be created for you. If you change your Python source code, it will create new byte code for you, and run that. We use the word *Python* to refer to either the source code, the byte code, the program that performs the conversion, and the language itself.



[5]The **user** is anyone uses or interacts with a computer. The **end user** typically refers to a customer who purchases a computer for themselves, or has a computer purchased for them by someone else such as the company they work for.
[6]I am not saying it would be easy.

Figure 1.3: In the traditional development process a program is written in a **text editor**, and **compiled** and **run** on the command line (the black squares). Errors are fixed one at a time by returning to the text editor and repeating the process (**debugging**). For some languages, or very complex programs, compilation is a complex, multi-step process that we represent here with the single command **`compile`**.



The Development Process

Python is a computer language. It has both a **syntax** and **semantics**. You When you speak in Python (actually you will write **text** that is called **code**), the listener (technically, the reader) is a computer. So Python is also **implemented** as a computer program that translates the the text that you write into stuff that the computer understands. When you type up a program in the Python language and tell the computer to interpret it, you are creating a new program. We will use the same word – Python – to refer to the language (the syntactical rules), the written language (the code or programs), and the computer program that translates your Python code into machine language or byte code (fig. 1.2). A special class of application programs – **development software** – are tools that allow or assist users to create these new computer programs.

In most traditional computer languages, you need to type your program into text files and save them. These files are then converted from source code (the text in the file) to executable code (programs the computer can run) by a process called **compilation** (fig 1.3). Compilation normally involves running one or more commands in the **command shell** of your operating system (such as **`cmd.exe`** or **`terminal`**; more on the command shell in chapter 4). Then the program can be run (aka, **executed**), also by typing a line in the command shell. The output is printed to the screen, one line at a time, in the command shell, or written to a file. This is the core of all scientific programs. Sometimes people like to add a nice **user interface** and an **icon** that people can click on. This turns the program in to an **application**. The design of user interfaces is a very deep subject and well beyond the scope of this text.

It is possible to develop Python programs using the traditional development process.

Because of the way Python was developed, you don't have to worry about the compilation step. When you try to run a program, the computer looks for compiled (byte code) first; if it is not there, it looks for source code and will compile it for you automatically. It will also compile the source code if it newer than the compiled code. This makes running Python code easier, though sometimes slower than other languages. Python also adds another development feature: the **Python shell**. The Python shell is a cross between a calculator mode and a miniature development environment (chapter 4). The **iPython shell** is a more advanced version of the Python shell, and allows direct access to many features of the command shell as well as high performance computing and macros in other languages (chapter 5). Finally, iPython notebooks allow the integration of text (such as LaTeX) and html, complete computer programs, output, and graphics into a single file that resembles a blending of MatLab and Mathematica interfaces (chapter 6).

Definition 1.2. Statement

The instructions in a computer program are called **statements**.

The down side of using a computer language is that the rules are **very precise**. There are no exceptions. These rules are called the **syntax** of the computer language.

If a rule is not followed, rather than being misunderstood, the program will likely either crash, or do something very unexpected.

Sometimes you will see an **error message** that seems totally unrelated to anything you did. This is because the computer interpreted things differently from the way you wanted. A comma replaced by a hyphen in spoken English will rarely throw off the listener, but it can lead to radically different interpretations by a computer.

There are two types of errors in computer programs: **syntactic errors** and **semantic errors**. The **syntax of a computer language** describes very precisely how to form statements the computer understands. If there is any deviation from the rules, even the smallest, the computer will recognize that there is an error, and it will tell you so. The **semantics of a computer program** describe what it is supposed to do. A semantic error that is not a syntactic error will not cause the program to halt. Instead, it will do something funny.

A common problem in Python is exponentiation. In many computer languages (such as Matlab), you can write exponentiation, such as 6^3, using the "carat" symbol, as in `6^3`. The statement `x=6^3` is syntactically correct in Python. However, it does not represent the number $6 \cdot 6 \cdot 6 = 216$. Instead you get the following somewhat peculiar result:

```
>>> print(6^3)
5
```

What you should have typed (to get 216) was `x=6**3`

```
>>>print(6**3)
216
```

The problem is that the carat operator is used in Python for the bitwise exclusive or operation.[7]. (Yes, exponentiation in Python uses the same syntax as FORTRAN!)

[7] See table 12.4: 6 xor $3 = (0110)_2$ xor $(0011)_2 = (0101)_2 = 5$.

Programming Paradigms

In the old days we used computer languages were classified by which **programming paradigm** or **style** they fit. As the number of different paradigms have grown, many of the languages have added features from other languages, so much so that the joke arose "the determined Real Programmer can write FORTRAN programs in any language."[8] Its actually quite difficult to even define the concept of programming paradigm, as (a) there are so many and (b) they overlap. Some of the better known paradigms (discussed very briefly below) include imperative, declarative, procedural, object oriented, and functional. Very few languages these days adhere to a single paradigm.

Definition 1.3. Imperative Language

A computer language or program is said to be **imperative** if control is based on a sequence of statements that change the **state** of the computer (e.g., values in particular memory locations[a]).

[a]Typically called variables.

FORTRAN and C are imperative languages. Imperative programming is the primary focus of this book. It is based on the algorithms, or the sequences of steps, used to solve a problem. When we assign a value to a variable, do an `if`/`elif`/`else` test, or execute a loop with a `for` or `while`, we are programming imperatively.

Definition 1.4. Declarative language

A language is said to be **declarative** if programs consist of problem specifications. Typical examples are Prolog, Modelica, and SQL.

A declarative program says **what** you want to do, not **how** to do it. A declarative program might say something like "Solve the matrix equation $\mathbf{Ax}=\mathbf{b}$ for $\mathbf{x}$" while an imperative program might say "The solution of $\mathbf{Ax}=\mathbf{b}$ is $\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}$."

Definition 1.5. Procedural Language

A computer language or program is said to be **procedural** if it can be broken down into one or more **procedures**[a] that can be **called** from a main program. A **procedure** is a sequential list of of instructions.

[a]Sometimes called functions or subroutines.

Procedural programs can actually be subclasses of other paradigms. Fortran and COBOL are both procedural.

When we define a function or class we are doing procedural programming. Object oriented programming is the ultimate in obsessive-compulsive procedural programming.

[8]Ed Post (1982)*Real Programmers Don't Use Pascal.*

Definition 1.6. Object Oriented Language

A computer language or program is said to be **object oriented** (OO) if programs consist solely of (a) **object**[a] definitions and (b) sequences of operations on those objects via special procedures defined to change the objects.[b]

[a]Sometimes called **data structures**; or **class**'s in Python (chapter 30).
[b]Called **methods**.

Languages that support object oriented programming include C++, Java, and Ruby. The main advantage of object oriented programming is data encapsulation: all operations on an object are hidden and done by the method associated with its implementation. A new implementation can (in theory) be substituted for an old one with no impact on other parts of the program design. This is a programming implementation of the concept of **systems analysis**, in which the design of a large project is broken up into different parts that can be handed out to different teams who work independently. A different team can be brought in to replace the others and only has to learn about the interfaces with but not the details of the other parts of the project.[9]

Definition 1.7. Functional Language

A computer language or program is said to be **functional** if computation is based on a sequence of function calls.

Typical functional languages are Haskell, Curry, APL, and Lisp. A functional program is a sequence of function calls. The functions do not necessarily represent objects or object modifiers; nor do they necessarily represent a sequential breakdown of operations. However, a functional program can be both object oriented and/or procedural. Functional programming is mostly useful for formally proving program correctness. However, many of the features of functional programming are useful and elegant and can be used in Python.

Which Paradigm is Right for Me?

It doesn't really matter what paradigm you believe is best or what is the coolest programming language. We take the algorithmic approach in this book because it is the most intuitive approach for mathematicians to follow.

The point of hacking is to get the job done. If you think you can get something done best by doing one part in SQL, another in R, a third in C++ and link then all together with Python and a bash front end, well then, by all means, do it that way.

[9]This has always been popular at NASA, ultimately leading to the destruction of two space shuttles and the death of everyone on board.

Chapter 2

A Tour of Python

This chapter contains a whirlwind overview of Python. It won't tell you how to create a program or even run a single line of code in Python – that will come in later chapters. Don't even expect to learn Python by reading it. The purpose is to introduce some concepts that you might not be familiar with, and to give you a sense of just what Python is.

It is nearly impossible to introduce a new programming language linearly to a student with no prior experience in a short period of time. It will seem like we are jumping back and forth and referring to new concepts before they are actually introduced. While we will try to keep this to a minimum, that cannot be completely avoided in a textbook like this. One of the reasons for this chapter is to get you used to some of those unexpected concepts before their time.

Keep in mind that our goal is not to learn about programming, but to learn to use computer programs to do math. If we had all the time in the world we could spend a whole semester just learning about all the ins and outs of objects and interfaces (we'll get to what those words mean; see chapters 29 and 30, for example) before we even thought about calculating a singular value decomposition (we'll get to that, too, in chapter 43). But if we want to spend *most* of our time *actually doing math*, then we want to dive into programming as quickly as possible. This is why we choose the hacking approach. **A programmer builds a product. A hacker solves a problem.** We want to solve problems, not sell products.[1] We will learn what we need, as quickly as possible, so that we can write (or hack out) working programs that do math.

Programs and Statements

In definitions 1.1 and 1.2, we introduced the concepts of **computer program** and **statement**. These definitions are very broad, and intentionally so. *In this book*, when we use the term statement, we will mean any single line of Python code. We will use the word program to mean any sequence of (one or more) Python statements that do something. It may be a Python function (chapter 18), a Python class (chapter 30), a module (a file that contains one or more Python programs), something that can be run from the command line (such as the `terminal` program in linux or OSX, or `cmd.exe` or `powershell` in, Windows; see chapter 4), one or more lines of code in the python shell (chapter 4), a file created by the `idle` programming environment, a single cell in an iPython notebook, or even an entire iPython notebook (chapter 6). It may be a single line of code.

[1] There is no rule that says we can't sell our solutions, or that an individual can't be both a programmer and a hacker, but programming is not our primary concern.

Python statements can be roughly grouped into two categories: **simple** and **complex**. Simple statements are things that can be stated on a **single line**. Complex statements can't fit on a single line.

Simple Statements

A simple statement is something that can be done in a single line of code. For example, in Python version 2, we can print the expression "Hello, World!" on the screen by including the line of code

```
print("Hello, World!")
```

in our program. In an **assignment statement** like

```
x = y-3*z+7
```

the value of the expression on the right hand side of the equation is computed, and then assigned to the variable on the left. In this example, the value of the variable **z** is multiplied by 3; then the result is subtracted from the value of the variable **y**; and finally, the number 7 is added to the previous result. The final result is stored in the variable **x**. The order in which the operations are computed is similar to the rules you probably learned in algebra class, but there are a few modifications because there are more operators than you probably expect. These rules are discusses in chapter 10. Assignments and other simple statements are discussed in more detail in chapter 11.

Numbers and Types

Python has three types of numeric representations: **int**, **float**, and **complex**. The **int** type is used for integers, which can take on any value between -2,147,483,648 and 2,147,483,647 on a 32 bit computer, and any value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 on a 64 bit computer.

The **float** type is used to represent anything that has a decimal point. These are called **floating point numbers**. Python floating point numbers can take on any value as large in magnitude as approximately 10^{308} and as small in magnitude as approximately 10^{-308}. Floating point numbers have approximately 17 digits of precision (see chapters 7 and 10 for more detail). A special complex number type (see appendix A for a review of complex numbers) does not really exist in Python. Instead, complex numbers are represented by a pair of **float**'s and the letter **j** (see page 73). For example, the number $2-3i$ would be represented by **2-3j**. The function **complex** can also be used to produce a complex number from a real number, or pair of real numbers.

In addition to the three numerical data types discussed, there is also a **boolean** data type. Boolean variables are used to represent expressions that have a truth value. A Boolean variable can take on one of two values, representing truth or falsehood. The special symbols **True** and **False** exist in Python to represent these values. In fact *any* numeric quantity can be used to represent a truth value; if the value is zero, the truth value is treated as **False**, and if it is non-zero, it is treated as true. Boolean expressions and data types are discussed in chapter 12.

Non-numeric data types include sequential data types like strings, lists, and tuplees. There are also objects called sets and dictionaries. Strings (data type **`str`**) are used to contain text like "My name is Fred" (chapter 16). A **`list`** is a comma delimited sequence of values enclosed in square brackets, like **`[1,2,3]`**. A **`tuple`** is similar to a list except that it is enclosed in parenthesis, as in **`(8,3,42,99)`**. The main difference between tuples and lists is that the lists are **mutable**, i.e., you can change the value of an element of a list after it has been assigned. Tuples are **`immutable`** (see chapter 15). A **`set`** is an unordered collection of unique items (chapter 17), analogous to a set in mathematics, Standard set manipulation operations like intersection and union can be applied to sets. A dictionary (data type **`dict`**) is organized like a printed dictionary: you can look up an entry by a key rather than a numerical index. For example, if you have a dictionary of phone numbers keyed by first name then **`phone["Fred"]`** would return Fred's phone number (chapter 25). Finally, you can always define your own data type with classes (chapter 30).

Lines and Indentation

Not all programming languages include the concept of lines; many of them ignore things like the carriage-return and line-feed characters in files or other hidden symbols in files that are used to represent the end of a line and the beginning of a new line. Python does not. Lines are significant in Python. The newline character in Python is **`\n`**, though in general you will not have to worry about this.

Not only are lines important, but so is spacing (indentation) within a line. Python implements **complex statements** (i.e., multi-line statements), to which we have alluded earlier, using **indentation**. A complex statement begins just before a block of indented lines, and ends with last indented line of code.

How you indent is just as important as **how far you indent**. When you use a word processor, you can indent paragraphs either with a tab key or some number of space characters. The tab key inserts a special character (represented by **`t`** in Python). Python sometimes expects a line of code to be **indented**. **When you indent your code, you must use space characters, and not the tab key.**

Sometimes a sequence of lines of code, called a **block** or **suite**, are supposed to be indented at the **same level of indentation**. This means the same number of blank spaces must begin each line of code within the block. The expression "level of indentation" means "number of blank spaces" at the beginning of the line. The amount of indentation does not matter, but it must be the same for each line in the block.

Sometimes we will have a block within a block that requires further indentation. If you have chosen to indent by 4 spaces, and have another block within your first block, then the inner block will be indented 8 spaces, like this:

```
1 line of unindented code
2 line of unindented code
3     first line of block
4     second line of block
5     third line of block
6         start of block within a block
7         second line of inner block
```

```
8           end of inner block
9       return to outer block
10      more of outer block
11      end of outer block
12  line of unindented code
13  line of unindented code
```

There is one block that starts on line 3 and ends on line 11. The block on line 6 ends on line 8; after it ends, the flow returns to the block from which the original code started, on line 9, which should agree with the indentation level on line 5. Examples of statements that require indentation are **if**, **while**, **for**, **with**, **try** and **def**.

Conditional Expressions and Statements

Conditional expressions and statements, discussed in chapter 12, perform an action based on the value of a Boolean (i.e., True/False) decision. Examples of Boolean decisions are questions like "is $x > 15$?", "is $(x + y)/2 < z^2$?", and "is $i + j$ even?"

The ternary operator (page 100), for example, can be used to define the step function

$$z = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{if } x \leq 0 \end{cases} \tag{2.1}$$

with a single line of code:

```
z = 1 if x > 0 else -1
```

The more traditional **if**/**elif**/**else** suites can be used to formulate more complex decisions trees. For example, a graduated income tax code that does not tax income below $1000, taxes at a 10% rate all income after the first $1000, up to $10,000, a 20% rate after that up to $30,000, and at a 30% rate on additional income, might be implemented as follows.

```
if income < 1000:
    tax = 0
elif income < 10000:
    tax = 0.1*(income-1000)
elif income < 30000:
    basetax = 0.1*(10000-1000)
    tax = basetax + 0.2*(income-10000)
else:
    basetax = 0.1*(10000-1000)+0.2*(30000-10000)
    tax = basetax + 0.3*(income-30000)
```

The **if** statement is discussed in chapter 12.

Functions, Classes, and Packages

Functions (chapter 18) and classes (chapter 30) give us a way to easily execute computer programs with a standardized interface. Functions describe how things are computed. Classes describe objects that functions operate on. It is not really necessary at this level to learn a lot about classes, since a sufficient number of data types are built into Python. As your hacking skills improve you will want to learn all about classes, because you can develop very elegant and efficient code using classes (e.g., object oriented programming), but that is the subject of other books. Functions, however, play a central role in scientific computing.

Functions have a form that looks like

```
somefunction(variable, variable, variable, ..., variable)
```

An example of a function that already exists in Python is the absolute value function, **abs(x)**. The function **abs(x)** implements the mathematical function $|x|$. Other typical functions that you will become familiar with are in the Python **math** package, such as **sqrt** (square root), **factorial**, and **atan** (arctangent) (see chapter 10).

Functions have both a domain and a range, just like in math. The domain variables are specified in the argument or parameter list, and the range values are given by the function itself. In math we say $f(x) = |x|$ maps the real numbers to the non-negative real numbers. In programming we say the function **abs(x)** returns a value that is the absolute value of **x**. The **return value** is in the range of the function. We obtain the return value by setting the function equal to a variable:

```
y=abs(x)
```

The value of **y** will contain the result calculated by the function **abs(x)**. It is very easy to define our own function in Python; consider the tax table introduced earlier. We could define a function **tax(income)** as follows:

```
def tax(income):
  if income < 1000:
    tax = 0
  elif income < 10000:
    tax = 0.1*(income-1000)
  elif income < 30000:
    basetax = 0.1*(10000-1000)
    tax = basetax + 0.2*(income-10000)
  else:
    basetax = 0.1*(10000-1000)+0.2*(30000-10000)
    tax = basetax + 0.3*(income-30000)
  return tax
```

We could save this in a file (a package) and then use it later as discussed in chapter 18. Whenever we wanted to use it we could call the function with the statement **y=tax(income)**, for whatever value of **income** we choose.

Loops

Computer programs use loops to repeat the same sequence of operations over and over again. There are four basic types of loops in Python: **for** loops, **while** loops, list comprehension, and generators. The first two types of loops are the most popular, and are present in nearly every mathematically oriented programming language in one form or another.

The **while** loop (chapter 13) is pretty similar to its analogues in other computer languages. The concept is this: while some boolean condition is true, continue to execute some suite of code. Each time the suite is executed (run) is called an **iteration**. As soon the boolean condition becomes false (even if it is false before the first iteration) stop and exit the loop. The following will add up the numbers 5, 10, 15, ..., 100 and print the sum, using a **while** loop.

```
total=0
n=5
while n <= 100:
   total = total + n
   n = n+5
print(total)
```

The **for** loop (chapter 19) iterates over a sequence and performs an action on every item on that set. The logic is this: for every item **x** in some sequence **S** perform the operations **f(x)**. To add up the numbers 5, 10, 15, ..., 100 using a for loop

```
total=0
for x in range(5,101,5):
    total = total + x
print(total)
```

One of the key differences between the two types of loops is that some sort of incrementation of the loop index is required in the **while** loop while it is done implicitly in the **for** loop. The incrementation is done by the statement **n=n+5** in the **while** loop, which takes the current value of **n**, adds 5 to it, and then points the variable to the new value.

To understand **list comprehension** (chapter 21) you need to understand Python list data types and mathematical set notation. Think of a Python list as a sequence of numbers separated by commas and enclosed in square brackets, such as

```
S=[2,4,6,8,9,11,15]
```

Suppose we want to generate a new list, **R**, that contains the square of every element in **S**. If these were sets, we could specify this mathematically by

$$R = \{x^2 | x \in S\} \tag{2.2}$$

We can write that almost literally in Python, replacing the $\in$ with the word **in**, the vertical line | with the word **for**, the expression x^2 with equivalent Python code **X**2**, and the curly brackets with square brackets:

```
 R=[x**2 for x in S]
```

Finally, we have generators. Generators are produced by any function that uses a **yield** rather than a **return** statement. This is analogous to lazy evaluation in functional programming (see chapter 29).

Libraries

There are many functions that extend the capability of Python, but which are not part of the basic definition of the language. To use these functions you must access a library with an **import** statement. The **import** must be executed before any function in the library is executed. (See chapters 11 and 18.)

There are two types of libraries: Python Standard Libraries, and other libraries. The only difference is that the Standard Libraries are defined in the language documentation at python.org and should be part of any standard Python installation. Other libraries may require additional software installation. The most common ways of installing these additional libraries are by using the program **pip**, although in some cases, a special install procedure is needed.[2]

Libraries are defined hierarchically and sometimes you will only need to install part of a library. The sub-parts are separated in the name by dots. Thus if you import the library **matplotlib.pyplot** that means you are importing only the **pyplot** sublibrary of **matplotlib**, and not all of **matplotlib**.

Installation

Python comes in two different versions:2 and 3. This book focuses on version 3. The current stable release[3] is version 3.7.0. Here is a brief summary of what you need to do to get started:[4]

1. **Install Python**.

   The latest version of Python can be installed from https://www.python.org/downloads/. Most Macs already have Python 3 installed.[5] Some computers may have both version 2 and version 3 already installed.

   Some users prefer a commercial system like Anaconda Python because they are easy to install and have dashboards for opening the program. Anaconda has the advantage that it can be installed for a single user without administrator privileges on most Windows systems.

2. **Install pip and jupyter.**[6]

[2] An index of which packages can be installed by **pip** is maintained at https://pypi.python.org/pypi. As of 23 Dec. 2015, 71626 packages were listed.
[3] As of 13 Aug 2018.
[4] This section discusses uses of the command shell, which is a different program in each operating system. You may want to look ahead and skim through chapter 4 which also discusses the command shell briefly.
[5] Linux users may prefer to use the latest version their repository. Note that many repositories, such as Ubuntu, tend to use earlier versions.
[6] This step should be replaced by instructions on the Anaconda website if you are using an Anaconda distribution.

There is a program called **pip** that is normally used from the command line to maintain and upgrade python distributions. All versions of python starting from python 2.7.9 and python 3.4 already have **pip** installed, so you will not have to install it yourself.[7][8]

In the command shell (the **terminal** program) on a Mac or the Windows command shell (e.g., **cmd.exe** or **powershell**), type the following:

```
python3 -m pip install --upgrade pip
python3 -m pip install jupyter
```

3. **Install the basic scientific libraries.**[9]

```
$ pip install ipython, numpy, scipy, matplotlib
```

If you are installed one of the scientific python packages (such as Anaconda) there may be different installation requirements. See the jupyter[10] and ipython[11] installation instructions.

Exercises

1. Locate the command shell program on your computer and open it. Type in the word **python** followed by enter. What happens? Is Python is already installed? If it is installed, what version is installed? Try using Python in "calculator" mode.
2. If Python is not installed on your system, install it.
3. Locate the command shell program on your computer and open it. Type in the word **ipython** followed by enter. Is iPython isntalled? Try out "calculator" mode here. Observe the differences between what you see here and what you saw in exercise 1.
4. If exercise 3 failed, then install ipython and jupyter on your system.
5. Locate the command shell program on your computer and open it. Type in the word **ipython notebook** and hit the return key to determine if the jupyter notbook is installed.
6. Upgrade your Python by adding **numpy**, **scipy**, and **matplotlib**.
7. Join Stack Exchange.[12] Identify the original author of each of the following quotes.

   (a) *Immature artists copy, great artists steal.*
   (b) *A good composer does not imitate; he steals.*
   (c) *Good artists copy. Great artists steal.*
   (d) *Good hackers imitate. Great Hackers copy.*

   It is common knowledge that when your professor assigns you a software project, you are going to do a web search for help. Most of the search results will either point to the Python documentation, which you may find incomprehensible, or to Stack Exchange, which you will find overwhelming. Join the community and at least find out the rules of searching and sharing so that you understand what you are looking at.

[7] If you are running Linux, you may prefer to perform updates through your software repository rather than **pip**; however, running **pip** should not cause any conflicts with the repository.
[8] It should not normally be necessary to install the program **setuptools** as it was in earlier versions.
[9] Anaconda users should instead use the Anaconda dashboard to install these packages.
[10] See http://jupyter.readthedocs.io/en/latest/install.html for details on installing the jupyter notebooks.
[11] See https://ipython.readthedocs.io/en/stable/install/index.html for details on installing ipython.
[12] See https://stackexchange.com/about.

Chapter 3

The Babylonian Algorithm

Suppose you want to find $\sqrt{2}$. Make a first guess, and call it x_1. Then if your guess is any good,

$$x_1 \approx \sqrt{2} \tag{3.1}$$

Therefore

$$x_1 \cdot x_1 \approx 2 \text{ or } x_1 \approx \frac{2}{x_1} \tag{3.2}$$

If x_1 is a reasonably good guess, then it is very close to $\dfrac{2}{x_1}$. Even if it is a poor guess, based on (3.2), we have no way to justify either of the following statements:

1) x_1 is a better guess than $2/x_1$.

2) $2/x_1$ is a better guess than x_1.

Since we have two equally good guesses for $\sqrt{2}$, why not take their average? It turns out that

$$x_2 = \frac{1}{2}\left(x_1 + \frac{2}{x_1}\right) \tag{3.3}$$

is always a better estimate of $\sqrt{2}$ than either x_1 or $2/x_1$, regardless of the value of x_1. Next, we apply the same argument to x_2. We arrive at the same conclusion. Repeating this process ad-infinitum,

$$x_3 = \frac{1}{2}\left(x_2 + \frac{2}{x_2}\right) \tag{3.4}$$

$$x_4 = \frac{1}{2}\left(x_3 + \frac{2}{x_3}\right) \tag{3.5}$$

$$x_5 = \frac{1}{2}\left(x_4 + \frac{2}{x_4}\right) \tag{3.6}$$

$$\vdots$$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right) \tag{3.7}$$

where each guess is better than the one preceding it.

We can stop when we reach our desired level of precision. For example, suppose we want to know the answer with a precision of six digits to the right of decimal point. Then we keep repeating this process until

$$|x_{n+1} - x_n| < 10^{-6} \tag{3.8}$$

The same argument applies to finding $\sqrt{a}$. The only difference is that instead of averaging x_1 and $2/x_1$, we average x_1 and a/x_1. The result is

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{3.9}$$

We will return to this iteration formula again and again throughout this book.

Bablylonian Algorithm

To find $\sqrt{a}$, let $x_1 = a$ and then iterate using

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{3.10}$$

Here's how we might write this up as an **algorithm**.

Algorithm 3.1 The Babylonian Algorithm.

input: a, tolerance ϵ
1: $x \leftarrow a$
2: **repeat**
3: $\quad x_{new} \leftarrow 0.5(x + a/x)$
4: $\quad \Delta \leftarrow |x_{new} - x|$
5: $\quad x \leftarrow x_{new}$
6: **until** $\Delta < \epsilon$
7: **return** x

The list of instructions above is not written in any computer language. It is an algorithm that is meant to be read by humans, and not by computers. It is written in a standard symbolic language that is midway between mathematics and computer programming. The format of the algorithmic language is designed to make it easy to translate directly into any computer language you like while retaining the basic mathematical content. Details like where you put semicolons and commas are omitted from this language.

Definition 3.1. Algorithm

An **algorithm** is a step by step procedure for performing a computation. It must contain an **identifiable entry point** and must **terminate after a finite number of steps**.

Here is what some of the symbols in algorithm 3.1 mean:

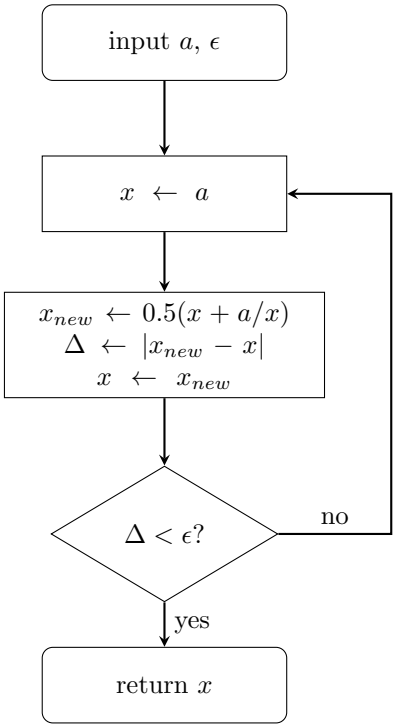
- **input**: gives the list of numbers that must be provided before you can begin the algorithm.
- $x \leftarrow a$ means replace the symbol x with the value of symbol a. The arrow is used to indicate which direction the copying goes: the symbol at the arrowhead is changed, while the symbol or expression at the foot of the arrow is not affected. In lines 3,

4, and 5, a computation is performed first, and the result of the computation is then copied into the symbol on the left.

- **repeat**..**until**: the entire sequence of steps is repeated, in order, over and over again. After each repeat, the expression after the **until** is evaluated. If it is true, then the repetition is terminated. If it is false, the sequence is evaluated again.
- **return**: gives the quantity that is returned. Everything else is invisible to the outside world except for the value returned.

Algorithms are often represented by flow charts. An flow chart of the Babylonian algorithm is shown in figure 3.1.

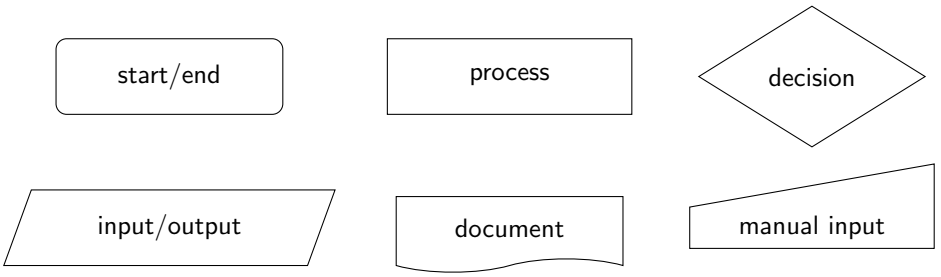
Figure 3.1: Flowchart for the Babylonian algorithm.



Flowcharts have a standardized format to make them easier to read at a glance (figure 3.2). **Start** and **stop** nodes are drawn as rounded rectangles or ovals. Usually start is at the top and stop is at the bottom but this is not always true. **Diamonds** represent **decisions**. Single or sequences of statements are collected together in rectangles which are called **process** boxes. Sequential flow is represented by the direction of the arrows, generally from top to bottom and left to right.[1] Arrows are generally aligned

[1]The top-to-bottom and left-to-right flow is often modified for typesetting purposes, so that an entire chart can fit into a smaller area. Of course loops require arrows pointing against the flow, so if there is anything really interesting going on there will be arrows in more than one direction.

Figure 3.2: Some standard flow chart symbols.



along horizontal and vertical paths and intersecting lines should be avoided. Additional standard boxes are defined for data, storage devices, documents, and manual entry.

Exercises

1. Estimate $\sqrt{20}$ to 7 significant figures using the Babylonian algorithm. Use your calculator or a spreadsheet to do the calculations. Compare with the correct answer. How many iterations do you need to get all 7 digits to match? How many additional digits do you get on each iterations?
2. Look up Newton's method in your freshman calculus book. Newton's method says that the root of a function $f(x)$ is given by successive iterations on the function $x_{n+1} = x_n - f(x_n)/f'(x_n)$. Show that the Babylonian algorithm can be derived from Newton's method.
3. Let $g(x) = \frac{1}{2}\left(x + \frac{2}{x}\right)$. Suppose $x = g(x)$. Find x.
4. Let $g(x) = \frac{1}{2}\left(x + \frac{a}{x}\right)$. Suppose $x = g(x)$. Find x.
5. Let $g(x) = \frac{1}{2}\left(x + \frac{a}{x}\right)$. For any fixed $a > 1$, show that there exists some number $K < 1$ such that $|g'(x)| \leq K$ in some neighborhood of $x = a$.
6. Suppose you use $g(x) = \frac{1}{2}\left(x + \frac{a}{x}\right)$ to generate a sequence of iterations $p_0, p_1, p_2, \ldots$ that are estimates for the root. Show that for each value of i there exists some number c_i such that

   $$|g(p_{i-1}) - g(p)| = |g'(c_i)||p_{i-1} - p|$$

   where $p = g(p)$ is called the fixed point of $g(x)$ (you found this in exercise 4).
7. Use the results of exercises 5 and 6 to show that

   $$|g(p_{i-1}) - g(p)| \leq K|p_{i-1} - p|$$

   Use the fact that $p_i = g(f_{i-1})$ to show that

   $$\begin{aligned} |p_{i-1} - p| &= |g(p_{i-2}) - g(p)| \\ &\leq |p_{i-2} - p| \\ &\leq K^2|p_0 - p| \end{aligned}$$

   where p_0 is your first guess. Use this to prove that the sequence $p_0, p_1, \ldots$ converges to p.
8. Write an algorithm for your typical morning. Include things like turning off the alarm clock, taking a shower, getting dressed, eating, making the coffee, feeding the dog, etc. Are there decisions that need to be made? Repeated processes (check to see if the dog needs to go out? is ready to come back in? did I turn the water off?).
9. Sketch a flow chart for filling up a bath tub. Things to consider: is the water hot enough? Did I put the drain stopper in? Is the tub full? Did I turn the water off?
10. Write down the steps and draw a flow chart for getting getting gas when you pay at the pump.
11. What is the big deal about the Babylonian algorithm? Why don't we just write down an expression for the the Taylor series for $f(x) = \sqrt{x}$ and add up terms until it converges numerically?

Chapter 4

The Python Shell

Shells and Stuff

You are probably most familiar with opening your apps by **point and click**. You find the right icon and click (or double click, depending on the operating system) to open it. Variations may include some level of navigation through the file hierarchy or menus (e.g., an application or start menu); a dock, task or launcher bar; or a heads-up display, in which you type an option key followed by the first one or two letters of the application's name. Typical applications you would open this way might include word processors, spreadsheets, presentation software, draw and paint programs, and web browsers. Nearly everything can be run this way, and for most users, this is sufficient. On hand-held systems, such as phones and tables, this is the only way that applications are intended to be run.

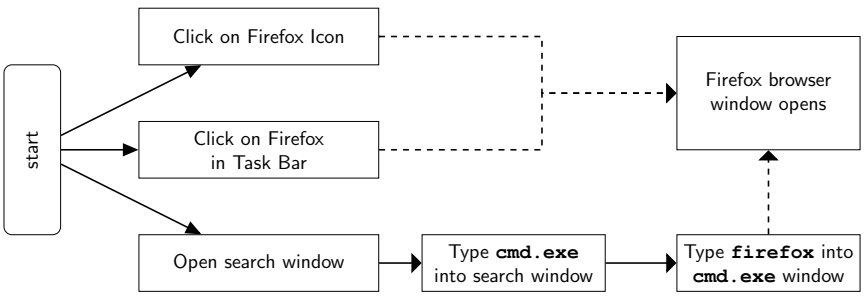
However, on full-fledged computing platforms, like laptop and desktop computers, there is another way to run your application. This is via a program called the **command shell**. The command shell is a special application; you can use it to tell your computer to run any other application by typing in a command on the keyboard.[1]

For example, to open the Firefox web browser, you can (see fig. 4.1):

1. Click on the **`Firefox`** desktop icon.
2. Click on the **`Firefox`** launcher, dock, taskbar, or start menu icon.
3. Open the command shell application and then type **`firefox`**, followed by Enter.

[1] Usually the command is just the name of the application itself, or a slight variation on the name of the application.

Figure 4.1: Three different ways to open the firefox web browser on Windows.



The command shell is basically just an empty palette for typing in commands that are sent directly to the operating system. All of the commands are in text, and must be entered from the keyboard.[2] The name of the command shell is operating system dependent.

Students who are new to programming often get confused by the following. There are **two distinct programs called shells**. These serve two different purposes, and **we are going to have to use both of them**.

- The **Command Shell**. In Windows, this is called **`cmd.exe`**. Use the search window: ⊞Win + S **`cmd.exe`**.

  On MacOS it is called **`terminal`**. Type ⌘ + space **`terminal`**.

  In Linux it is also called **`terminal`**. Type ctrl + alt + t.
- The **Python Shell**. The Python Shell should **only** be opened by entering a command in the Command Shell. We will talk about three different types of Python Shell in this text (fig. 4.2).
  - ▶ The traditional Python Shell (in this chapter).
  - ▶ The iPython Shell, a more advanced version of the Python shell (chapter 5).
  - ▶ IPython Notebooks, which provide the ability to include whole programs, text, output, and graphics in a single file (chapter 6).

  The Python language used is identical, no matter which of these interfaces you choose to use.

Doing it in Python

Here's an overview of how you can use Python without using the Python shell at all:

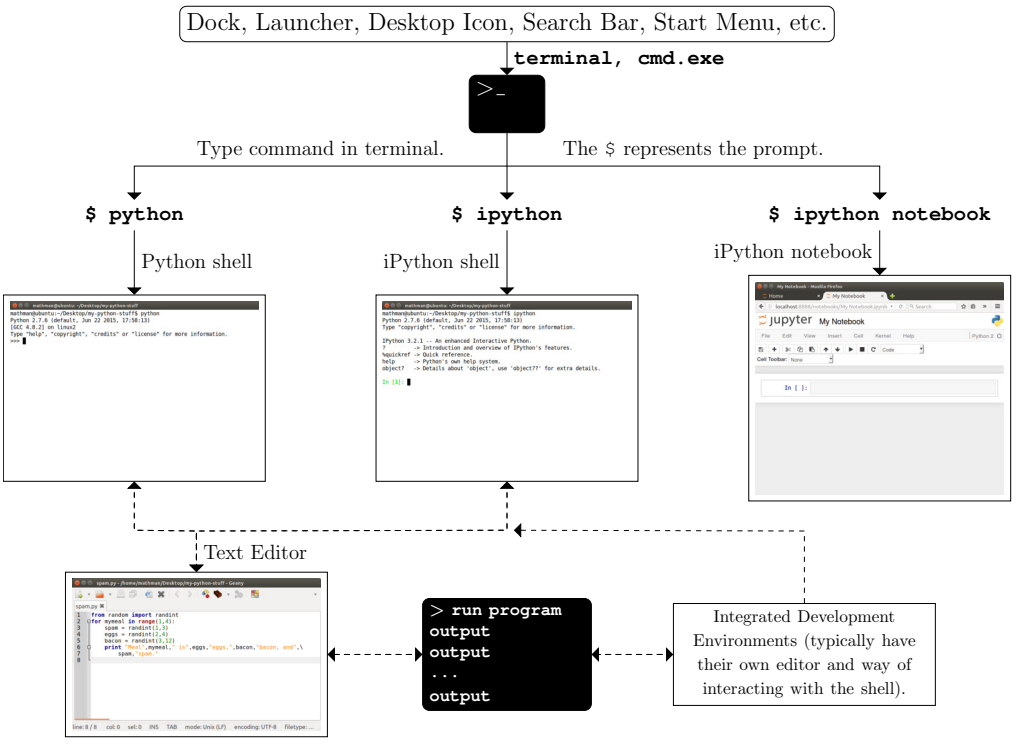
- Write a program in a text editor and save it disk. Then run the program from the command shell. You have to debug it iteratively (like in figure 1.3).

Here's an overview of what you can also do using the Python shell:

- Run single lines of Python code, like a very fancy calculator.
- Type (and run) multiple lines of Python code (even a whole program). This is somewhat tedious, since if you make even a single mistake, you have to type the whole thing in all over again.
- Write programs in a text editor. Save them to disk. Load and run the them in the Python shell.

[2] For windows users, a good place to learn about the command shell quickly is *The CLI Crash Course* by Zed Shaw, which you can read on line for free at http://cli.learncodethehardway.org/. The Mac OS uses a variation of the linux bash command shell, but one place to start learning about it is the book by Joe Kissell, *Take Control of the Mac Command Line with Terminal, 2nd Ed.* For Linux users I suggest Machtelt Garrels' *Bash Guide for Beginners*.

Figure 4.2: Comparison of the different ways of invoking Python that are discussed in the text. The black rectangles with curved corners represent command shells.



Furthermore, many editors and programming environments have direct connections to the Python shell. Examples include:

- **Idle** - this is a mini-IDE[3] that includes a text editor designed for Python and its own version of the Python Shell. You can run the program being edited with a single keystroke, usually F5.
- Many text editors, such as **Geany**[4], have direct connections to the shell.
- All major IDE's have Python support; examples include **Eclipse**[5] (e.g., using the **PyDev**[6] plugin); **Komodo**[7]; and **XCode**.[8]

[3]Integrated Development Environment. The name **idle** refers to the actor Eric Idle, who was in the original Monty Python TV show. There is another Python mini-IDE calle **eric**.
[4]See https://www.geany.org/. In **Geany**, using build > execute or F5 will open a console that automatically runs the program.
[5]https://eclipse.org/
[6] See http://www.pydev.org/; on all operating systems.
[7]See http://komodoide.com/; on all operating systems.
[8]See https://developer.apple.com/xcode/ide/; available on MacOS only. XCode is the "official" Apple-blessed coding environment for MacOS.

Introductory Tutorial

How might we implement the Babylonian algorithm Python?

1. Open the Command Shell, as described on page 22. When the command shell opens, type the following to open a Python Shell, followed by enter:

```
python
```

   The first thing you should notice is that the **Command Shell Prompt** (the character(s) on the far left of the line that tell you the computer is ready for input) will have have been replaced by the **Python Prompt**.

   - The **Command Shell Prompt** is either "**>**" (on Windows):

```
>
```

     or "**$**" (on Mac OS or Linux):

```
$
```

     Sometimes it will have the current user name and directory written in front of it:

```
username@computername:path/to/home/directory$
```

   - The **Python Prompt** is always "**>>>**"

```
>>>
```

   You can always identify the Python Shell by the prompt: **>>>**.

   Before the first Python Prompt is written to the screen, you will see some information about the current version of Python installed on your system.

```
mathman@ubuntu:~/Desktop$ python3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
   information.
>>>
```

   Observe that you are still running the Command Shell, but the Python Shell has taken over control of the Command Shell window.

   To leave the Python shell at any time (and return to the Command Shell), type ctrl+d (or **exit(0)** followed by enter) at the Python Prompt.

2. Type the following code into the Python shell. **Do not type in the >>> prompts**. After each line, type enter.

```
>>> x1=2
>>> x2=0.5*(x1+2/x1)
>>> print(x2)
```

A new prompt should appear after each line, except that something different happens after the **print**. When you press Enter after the **print** statement, the number **1.5** should be returned by the computer. Since there is no prompt before the **1.5**, you know that the computer printed this number and you did not type it in yourself. The line after the **1.5** has a prompt, telling you that you are free to enter another command to Python.

```
1.5
>>>
```

3. Open your favorite text editor. Windows comes with **Notepad**, and MacOS comes with **textedit** already installed. You can use these, but each has its own idiosyncrasies. If programing is new to you, I suggest you start with one of these two built in editors. If you are more adventurous, or when you get frustrated with them due to their lack of features, then you may want to install something better.[9]

   - If you use **Notepad** on Windows, make sure that when you save your file, it gets saved as a Python code text file. To do this, select Save as type » text files and that your file name ends with the extension **.py**
   - If you use **textedit** on the Mac, the default file format is RTF; you want to make sure your file is marked as a Python code text file. To do this, before you save your file, either select format » make plain text or ⇧+⌘+T. Make sure your file name ends with the extension **.py**.

   Open up a new file and type in the following code.

```
x1 = 2
x2 = (x1+2.0/x1)*0.5
print("x2=",x2)
x3 = (x2+2.0/x2)*0.5
print("x3=",x3)
x4 = (x3+2.0/x3)*0.5
print("x4=",x4)
x5 = (x4+2.0/x4)*0.5
print("x5=",x5)
x6 = (x5+2.0/x5)*0.5
print("x6=",x6)
x7 = (x6+2.0/x6)*0.5
print("x7=",x7)
```

4. Save the file as **babylalg.py**, e.g., using the menus file » save as **babylalg.py**.

[9] Popular editors that are easy to get started with are **geany** (available on all platforms, https://www.geany.org/); **textwrangler** (MacOS only, http://www.barebones.com/products/textwrangler/); **Notepad++** (Windows only, https://notepad-plus-plus.org/); and many IDE's.

5. Open a new command shell (not a Python shell), and verify that your file exists in the local folder.

   - In **Windows**:

     ```
     $ dir babylalg.py
     ```

   - In **MacOS** or **Linux**:

     ```
     $ ls babylalg.py
     ```

   This asks the computer to list of all the programs in the local folder (directory) named **babylalg.py**. You should see one line, which looks something like this:

   ```
   babylalg.py
   ```

   If you do not see your program listed then you are not in the appropriate folder. Go back and figure out where you saved your program and open a new command shell there, or find the folder where you did save the program to and then type

   ```
   $ cd PATHNAME
   ```

   in the command line. The same command will work on all operating systems. You need to replace **PATHNAME** with the full path of the folder where you saved your file. Repeat the previous paragraph until you succeed.

6. When you have found your file, type the following:

   ```
   $ python babylalg.py
   ```

   As soon as you press enter, the computer will spit out several lines, like this:

   ```
   x2= 1.5
   x3= 1.41666666667
   x4= 1.41421568627
   x5= 1.41421356237
   x6= 1.41421356237
   x7= 1.41421356237
   ```

   If your output looks something like this, then you have succeeded in running your program from the command shell. We will see how to make this algorithm much more elegant in future chapters.

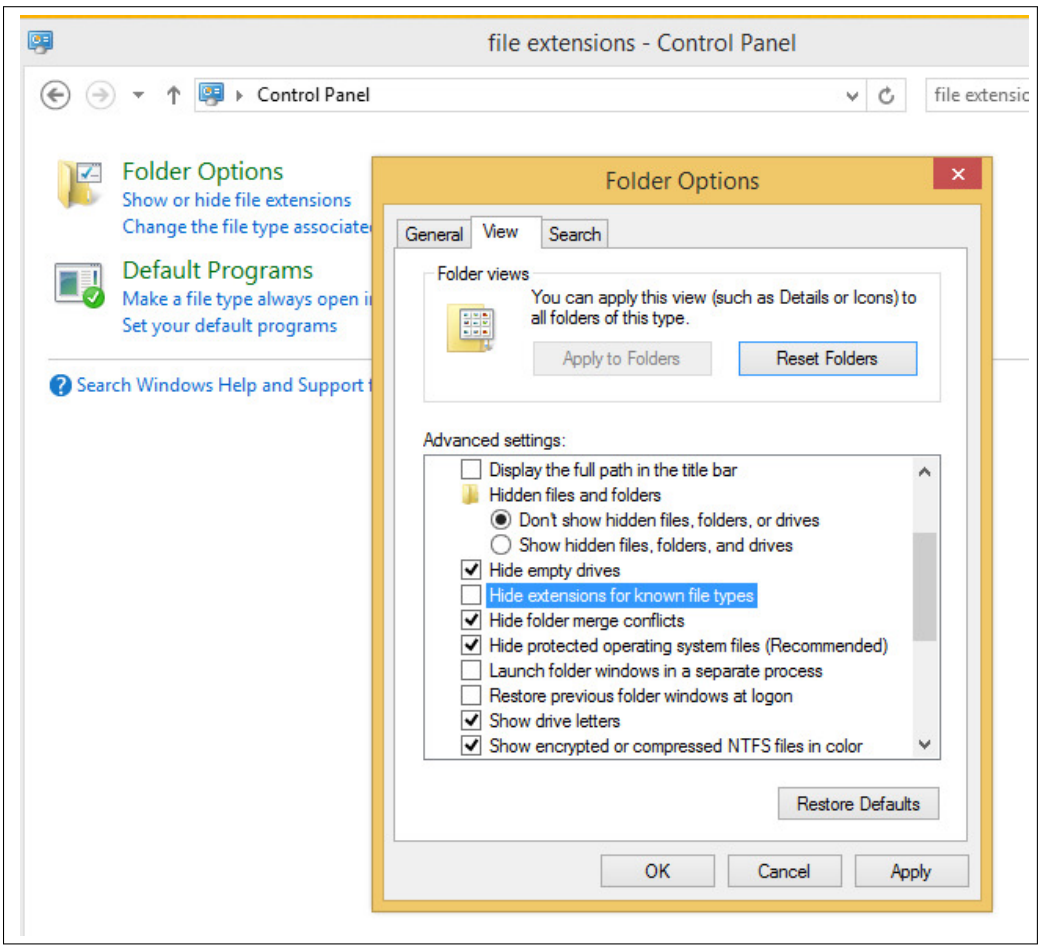
We will see in the next chapter that using the iPython shell instead of the Python shell makes the process of code development somewhat easier. This is because iPython has some more advanced features such as direct access to the command shell without having to open up a new window. In chapter 6 we will introduce iPython notebooks, which combine input, output, text, programs, and complete editing capability in a single interface.

File Extensions for Windows Users

In all versions of Windows operating systems, the portions of the file name following the period are normally hidden from the user. This is because Windows thinks it is smart enough to figure out what those extra letters means. Sometimes it does, and sometimes it doesn't. If you want to know which files are Python programs, you will need to see the file extension, which must be **.py**. To fix this, do a search » file extensions. This will bring up the control panel for folder options. Open the menu option Control Panel » Folder Options » View » Hide extensions for known file types. Make sure the box is not checked off. Then click on Apply.

File extensions are normally shown on Mac and Linux systems so no additional modification is required.

Figure 4.3: Uncheck Hide extensions for known file types then Apply.



Exercises

1. Explain the steps needed to open the python shell on your computer. What is the difference between python and idle?
2. Compare and contrast the python shell and the command shell.
   (a) List at least three things you can do in the command shell that you can not do in the python shell.
   (b) Describe something you can only do in the python shell.
   (c) Explain how to the open the python shell and how to open the command shell.
3. How is a command line program (or command shell program) different from one you write and run in the Python shell? Explain how you write a command line program on your computer?
4. What is the difference between an interactive program and a command line program? A command line program and an application?
5. Figure out how to open up the command shell on your computer. Open the command shell and from the command shell do each of the following:
   (a) print the name of the current directory (folder)
   (b) print the names of all the files in the current directory
   (c) print a list of the names of all the files in the current directory with only one name per line and nothing else on the line
   (d) print a list of the files in the current directory showing the file size, creation date, and file protection
   (e) without changing the current directory, print the name of the parent directory
   (f) print a list of all files of extension **`.png`** in the current directory. If there are no **`.png`** files there, find a picture on the internet, and save one to the current folder using your web browser and then repeat this task.
   (g) change your current directory to the parent directory
   (h) show the new directory
   (i) set your directory to your home folder
   (j) create a new folder called **`python-homework`**
   (k) set your directory to **`python-homework`**
   (l) open the Python3 shell by typing **`python3`** in the command line
6. In the Python shell,
   (a) Assign a values of 7 and 10 to variables **`p`** and **`q`**
   (b) Calculate and print out $p + q$, $p - q$, pq and p/q. For the quotient, calculate the result both using integer and floating point division.
   (c) Find 10 mod 7 and 7 mod 10 using Python
   (d) Type in the line **`import math`** and then hit the enter key
   (e) Find $\sqrt{2}$ by typing **`sqrt(2)`**
7. Type in the code for a single calculation of the Babylonian algorithm for $\sqrt{7}$. Let $x_0 = 7$. Find x_1. Repeat several calculations. Try to get the algorithm to converge to 7 places.
8. Create a text file that contains the code for ten iterations of the Babylonian algorithm for $\sqrt{37}$. Have it print the output after each iteration, as we did on page 25. Save the file and run it from the command line.
9. It is traditional in programming classes for your first program to be one that prints out the string "Hello World!" to the screen. Write a "Hello World!" program and save it to a file **`hello.py`**. Run it from the command line.
10. Familiarize yourself with the official online documentation.[10]
11. Read the official Python tutorial.[11]

[10] You can the documentation at https://www.python.org/.
[11] The tutorial is at https://docs.python.org/2/tutorial/.

Chapter 5

The iPython Shell

IPython[1] is a command shell that extends the functionality of the usual Python shell.[2] The Python programs that you write and execute using iPython are identical to the ones that you can write and execute in the ordinary Python shell. There is no need to change your code - the Python language is the same. It is just a different development environment. The main difference is that iPython provides support for high performance computing, in-line graphics, system commands, and a notebook interface (about which we will have more to say in chapter 6).

The iPython shell is invoked from your **command shell** (see page 22) as follows:

```
$ ipython
```

The start-up message is a little different from the start-up message in the standard command shell (compare with page 24):

```
mathman@ubuntu:~/Desktop$ ipython3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from math import sqrt

In [2]: sqrt(5)
Out[2]: 2.23606797749979

In [3]:
```

However, the most notable change is that the command prompt ("`>>>`" in the "traditional" Python shell) has be replaced by the sequence of prompts "`In[1]:`", "`In[2]:`", "`In[3]:`", etc, in the iPython shell. Furthermore, if the command has a return value, as was the case with the command on `In[2]`, but not `In[1]`, above, there is a correspondingly labeled `Out[2]` line.

[1] Fernando Pérez, Brian E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science and Engineering*, 9(3):21-29 (2007).; http://ipython.org.

[2] This chapter can be safely skipped by readers who are not interested in using iPython or who will not be using either iPython or iPython notebooks. Students who are new to programming can skip the details on a first reading.

What is Going On?

Its a little bit easier to see what is going on in iPython than in the traditional Python shell because things are clearly labeled. The Python (or iPython) shell has a continuous evaluation loop that is always waiting for you to input an instruction. Whenever it is ready and waiting for you to enter a new instruction, iPython will prompt you with

```
In [1]:
```

The traditional cell will replace the "`In [1]:`" with "`>>>`," and the input line won't be clearly set off from the previous line. When we get to notebooks in the next chapter, the reset of the blank input cell will be outlined (like it is here) and filled up in light gray to set it off from the white background.

When you type something into the input line (or cell in the notebook), Python will attempt to interpret it. When Python is done, it will do one of the following:[3]

- **Nothing**. This is misleading because sometimes there is a side effect, such as changing the value of a variable. The following sets `x` to `5`. By sending you the blank prompt at input 2, it may seem like nothing has happened.

```
In [1]: x=5

In [2]:
```

- **Print something**. The `who` command prints out the currently defined variables.

```
In [1]: x,y,z=5,10,15

In [2]: who
x  y  z

In [3]:
```

- **Return a value**. This will be placed in an output cell.

```
In [1]: 5+7

Out[1]: 12

In [2]:
```

  Anything which goes into an output cell can be assigned to a variable; then nothing will be printed out.

[3] One may argue that it can do other things like issue a pop up menu asking for input, write a file, or start a sub-process. However, that is not correct, as these are things that Python *can do*, and not things that Python does to signal that *it is finished*.

Magic Commands

Of course, if it were just a matter of adding line numbers, there would be little point to having a whole different command shell. But there's more. Much of it comes from the use of **magic commands**. These are special commands that the iPython shell recognizes that provide extra functionality. Any line that starts with a percent symbol (`%`) is treated by iPython a magic command. There are two types of magic commands:

- **Line magic commands**: The entire command, including any arguments it may have, is on a single line. For example, the following first executes the system command **`%pwd`**, whose return value is a string with the name of the current directory. This is followed by a change directory command **`%cd`**, to change to the folder **`Desktop`** withing the current working directory. This does not have a return value, but does automatically print the value of the new directory after the working directory is changed.[4]

```
In [3]: %pwd

Out[3]: '/home/mathman'

In [4]: %cd Desktop

        /home/mathman/Desktop
```

  Line magic commands are summarized in table 5.1.

- **Cell magic commands**: These include multiple lines in the current indentation block. A cell magic begins with a double percent (`%%`). If necessary, extra prompts "`....:`" on each line are automatically added by iPython until the input is complete. For example, we can use **`%%script`** to write a **`bash`** (or other scripting language that is present on your computer) and run it.[5]

```
In [5]: %%script bash
          ....: for i in 1,2,3; do
          ....:   echo $i
          ....: done
          ....:

        1,2,3
```

  Cell magic commands are summarized in table 5.2.

[4] It is no accident that the syntax of these commands is similar to the corresponding Linux commands.
[5] IMPORTANT NOTE: This is for illustration only; beginners are not expected to know about scripting, or what **`bash`** is. The take-away message is that sometimes it takes multiple lines to do things, and we have to use a different syntax for that.

Table 5.1. iPython Magic Commands (1 of 5)

Command[a]	Description
%alias	Makes an alias (nickname) for a system command. **%alias myalias command** defines **myalias** as an alias for **command** **%alias** List all aliases
%alias_magic	Makes an alias (nickname) to a magic command.

%autocall	Makes a function callable without parenthesis.
	%autocall 0 Disable auto-call.
	%autocall 1 Enable auto-call.
	%autocall Toggle auto-call.
	Example:
	`In [38]: %autocall`
	`Automatic calling is: Smart`
	`In [39]: sqrt 36`
	`-------> sqrt(36)`
	`Out[39]: 6.0`

%automagic	Toggles use of % symbol for magic key (see page 37). **%automagic 0** % is required. (or: **off**, **False**) **%automagic 1** % is not required. (or: **on**, **True**) **%automagic** Toggle auto-magic.

%bookmark	Set, list, or remove a bookmark. These are just like the bookmarks that you set to folders or urls on your desktop or web browser.
	%bookmark name folder bookmark **folder**.
	%bookmark name bookmark current folder.
	%bookmark -r name remove bookmark.
	%bookmark -r remove all bookmarks.
	%bookmark -l list all bookmarks
	`In [1]: %bookmark spam "/home/mathman/Desktop"`
	`In [2]: cd spam`
	`(bookmark:spam) -> /home/mathman/Desktop`
	`/home/mathman/Desktop`

%cd	Change current working directory. **%cd directory** change to given directory. **%cd -** change to last directory. **%cd -n** change to nth last folder (n=integer). **%cd -q** quiet mode (don't print folder name).
%colors	Select color scheme. Options are **NoColor**, **Linux**, **LightBG**.

[a] See http://ipython.readthedocs.io/en/stable/interactive/magics.html for more details.

Table 5.1. iPython Magic Commands (2 of 5)

Command	Description
%config	Set configurable traits. **%config** — List configurable classes. **%config cname** — List configurable traits in the class **cname**. **%config cname.trait=value** — Set the desired trait to the specified value.
%debug	Interactive debugger.
%dhist	Print history of last n directories visited.
%dirs	Returns the current directory stack.
%doctest_mode	Toggles interface to look like normal Python shell and back.
%edit	Opens an external text editor. Uses the default text editor on your operating system. Selection priority: 1. As listed in the iPython configuration file (see page 41). 2. The value of the **$EDITOR** environment variable. 3. Either **vi** (linux/max) or **notepad** (windows). The **%edit** magic is not available in iPython notebooks.
%env	Access environment variables. Environment variables are user-configurable variables used by your operating system to define the way your computer works. Most users can get by with never changing any of them directly. Advanced users frequently find it useful to change or access th system configuration. **%env** — List all environment variables and their values. **%env name** — Show value of **name.** **%env name value** — Set value of variable **name**.
%gui	IPython has hooks for most standard graphical user interfaces that advanced users would want to work with. This magic command allows users to toggle GUI event loop integration. The following options are available: **gtk**, **gtk3**, **qt**, **qt4**, **qt5**, **tk**, **wx**.
%history	Print history. Various options may be combined. An optional **range** of cells may follow, as in: **%history -n 17-20**. **%history -n** — Print line number with input. **%history -o** — Print output. **%history -p** — print classic Python prompt "**>>>**." **%history -f F** — Print to file **F**.
%install_ext	Installs an iPython extension from a specified URL. Normally extensions are loaded used **pip**, like any other Python extensions. An iPython extension can be loaded while running using the **%load_ext** magic command.

Table 5.1. iPython Magic Commands (3 of 5)

Command	Description
`%killbgscripts`	Kills all background processes started by `%%script`.
`%load`	Loads a code module into the front end.
`%logoff`	Temporarily stop logging that was started with `%logstart`.
`%logon`	Restart logging that was temporarily stopped by `%logoff`. Does not initialize logging, which must be started with `%logstart`.
`%logstart`	Starts logging the first time in current session. `%logstart` Begin logging of input. `%logstart -o` Include output in log file. `%logstart -t` Include time stamps in comments. The `-o` and `-t` options may be combined and may be optionally followed by an output file name and output mode for the output file. The default file name `ipython_log.py`. The mode is: `append` Append to the end of existing log file. `backup` Rename existing `file` to `file`~ and create new file. `global` Appends to existing file in home folder. `over` Overwrites existing files. `rotate` Creates new file names `file.1`~, `file.2`~, etc.
`%logstate`	Reports the state of the logging system.
`%logstop`	Stop logging and close the log file.
`%ls`	List all files in current working directory. Standard gnu linux options. Compatible options can be combined. `%ls -a` Show all file names. `%ls -d` Just show sub-directories. `%ls -l` Tabular format with additional information about each file. `%ls -R` Recursively list contents of sub-directories. `%ls -s` Show file size.
`%lsmagic`	List all magic commands.
`%macro`	Define a macro.
`%magic`	Print summary of the magic system.

Table 5.1. iPython Magic Commands (4 of 5)

Command	Description
%matplotlib	Initializes plotting system to work interactively. To display plots inline in iPython notebooks, use **%matplotlib inline** Other GUI backends may be available in some OS's. **%matplotlib -l** List all backends. **%matplotlib** GUI Set backend to GUI, if available. Typical values: **pyglet**, **osx**, **qt5**, **qt**, **glut**, **gtk**, **gtk3**, **tk**, **wx**
%notebook	Export current history to a notebook. A file name must be specified: **%notebook spam.ipynb**
%page	Pretty print an **x**: **%page x**
%pdef	Print the call string of **x**: **%pdef x**
%pdoc	Print the document string of **x**: **%pdoc x**.
%pfile	Print the file where **x** is defined: **%pfile x**
%pinfo	Print information about **x**: **%pinfo x**.
%popd	Pop directory off top of stack and change to that directory.
%pprint	Toggle pretty printing.
%precision	Set floating point precision for pretty printing.

%prun	Profile current session or statement.
	`In [15]:  %prun`
	`     2 function calls in 0.000 seconds`
	`     Ordered by:  internal time`
	`     ncalls tottime percall cumtime percall ...`
	`          1   0.000 0.000 0.000 0.000 ...`
	`          1   0.000 0.000 0.000 0.000 ...`

%psearch	Search for item in a namespace. **%psearch** pattern type pattern is search string. Wild cards may be specified with "*". type is **function**, **integer**, **string**, etc.
%psource	Source code for **f**: **%psource f**.
%pushd	Push current directory onto the directory stack and change the working directory.
%pwd	Print the full path name of the current working directory.

%pycat	Concatenates the specified file with Python syntax highlighting. Similar to the **cat** utility in linux.
	`In [18]:%pycat spam.py`
	`from random import randint`
	`for mymeal in range(1,4):`
	`     spam = randint(1,3)`
	`     eggs = randint(2,4)`
	`     bacon = randint(3,12)`
	`     print("Meal",mymeal," is",eggs,"eggs,",`
	`       bacon,"bacon, and",spam,"spam.")`

Table 5.1. iPython Magic Commands (5 of 5)

Command	Description

`%pylab`	Clutters up the namespace. Equivalent to:
	`import numpy`
	`import matplotlib`
	`from matplotlib import pylab, mlab, pyplot`
	`np = numpy`
	`plt = pyplot`
	`from IPython.display import display`
	`from IPython.core.pylabtools import figsize, getfigs`
	`from pylab import *`
	`from numpy import *`

`%quickref`	Display a reference card.
`%recall`	Recall a command by input number.
`%rehashx`	Updates the alias table with executables in `$PATH`.
`%reset`	Remove all user-defined names. Example page 39.
`%run`	Runs an external file or script, e.g., `%run myprogram.py`. Example page page-magic-run.
`%set_env`	Sets an environment variable.
`%sx`	Run shell command, e.g., `%sx whoami`. Example page page-magic-run.
`%system`	Same as `%sx`.
`%time`	Print cpu and wall clock time. Example page 37.

`%timeit`	Time execution of a Python statement or line of code.
	`In [33]: %timeit sqrt(36)`
	`The slowest run took 41.72 times longer than the fastest. This could mean that an intermediate result is being cached`
	`10000000 loops, best of 3: 74.3 ns per loop`

`%unalias`	Removes an alias.
`%unload_ext`	Unloads an extension. Some extensions cannot be unloaded.
`%who`	Prints all variables currently defined. Example page 39.
`%who_ls`	Returns all variables as a sorted list. Example page 39.
`%whos`	Prints a formatted table of all variables, their times and assigned values. Example page 39.
`%xdel`	Removes a variable: `%xdel y`. Example page 39.

Automagic Mode

By default, **automagic** mode is enabled in iPython. This means that you normally won't need to use the leading "%" character unless the magic command becomes **shadowed** by a command with same name. For example, the magic command **%time** prints out some time information for the current session. If you have not defined a local variable **time**, then by entering **time**, you get the output of the **%time** magic command:

```
In [6]: time

        CPU times: user 3 ns, sys: 1 ns, total: 4 ns
        Wall time: 7.87 ns
```

As soon as you define a local variable **time**, the only way you can access the magic command is by using the "%."

```
In [7]: time="12:00 Noon"

In [8]: time

Out[8]: '12:00 Noon'

In [9]: %time

        CPU times: user 4 ns, sys: 2 ns, total: 6 ns
        Wall time: 45.1 ns
```

When we defined the variable **time** in **In[7]**, it **shadowed** the magic command. This means it prevents it from being seen by the front end.

Magic commands are always accessible by affixing their leading "%." If **automagic** mode is enabled, the command can be accessed either with or without the "%," unless it is shadowed. If **automagic** is disabled, then all magic commands can only be invoked using their leading "%." To toggle **automagic** mode, use the **%automagic** magic command alone. Alternatively, you can explicitly set **%automagic** equal to **0**, **off**, or **False** (to turns it off); or to **1**, **on**, or **True** (to turn it on).

System and Folder Access

There are magic commands that will show you the name of the current working directory, its contents, and change the directory. The syntax of these commands is identical to their Linux form. To print the current directory, use **pwd**; to change to another directory, use **cd** (see page 31). The double-dot notation ("..") represents the next directory up.[6] You can list all of the files in the current folder with the **%ls** magic.

Any system command can be executed with the **run** magic. For example, suppose that you type the following Python program into a text editor, and save it as **spam.py**.

[6]So if your current directory is **/fred/pythoncourse/projectone** then **..** refers to **/fred/pythoncourse**.

```
from random import randint
for mymeal in range(1,4):
     spam = randint(1,3)
     eggs = randint(2,4)
     bacon = randint(3,12)
     print("Meal",mymeal," is",eggs,"eggs,",bacon,"bacon, and",\
         spam,"spam.")
```

The you can invoke **spam**[7] like this:

```
In [10]: run spam.py
```

```
Meal 1  is 4 eggs, 5 bacon, and 2 spam.
Meal 2  is 3 eggs, 11 bacon, and 3 spam.
Meal 3  is 2 eggs, 8 bacon, and 3 spam.
```

Any system or shell command can be run with the **sx** magic. For example, the Linux **lsblk** returns a table of information about the file system.[8] You can run this command from inside iPython (if you are running on Linux; you would only be able to run Windows commands on a Windows computer and MacOS commands on a Mac):

```
In [11]: sx lsblk
```

```
Out[11]: ['NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT',
          'sda      8:0    0   256G  0 disk ',
          '\xe2\x94\x94\xe2\x94\x80sda1   8:1    0   256G  0 part /',
          'sr0     11:0    1  1024M  0 rom  ',
          'sr1     11:1    1  1024M  0 rom  ']
```

Managing Your Workspace With Magics

Many magics are defined to manage and log the history of what you have done. You can access any of the previous inputs and outputs treating the variables **In** and **Out** as indexed variables, e.g., **Out[29]** will return the value of output cell 29. You can also log all or part of your session to a file. These commands are summarized in table 5.1.

Other magics that can be particularly useful during a session include:

- Magics to help you learn remember magic:
  - **%magic** describes the magic system.
  - **%lsmagic** lists all magic commands.
  - **%quickref** prints out a quick reference card to the screen.

[7] Note that this is just for illustration. We will not introduce for loops until n chapter 19, and random numbers until chapters 31 and 32, so don't expect that you should need to be able to produce this code right now in order to continue to understand the book. If you actually decide to type this in and run it, expect the literal numbers printed to be different, because of the random number generator.

[8] You don't have to know anything about **lsblk**, file system details, or Linux to continue reading this book. The point of this example is that you can run even some very esoteric and mysterious looking commands from iPython.

- Magic commands that tell you the variables defined in the current session:
  - **%who** prints the variables to the screen:

```
In [12]: who
```

```
bacon  eggs  mymeal  randint  spam  time
```

  - **%who_ls** returns the variables as a list:

```
In [13]: who_ls
```

```
Out[13]: ['bacon','eggs','mymeal','randint', spam','time']
```

  - **%whos** prints a table of the variables:

```
In [14]: whos
```

```
Variable   Type              Data/Info
-------------------------------------
bacon      int               8
eggs       int               2
mymeal     int               3
randint    instancemethod    <bound method Random.r<...>>
spam       int               3
time       str               12:00 Noon
```

- Magics to remove variables:
  - **%xdel** removes a single variable.

```
In [15]: xdel eggs
```

```
In [16]: who_ls
```

```
Out[16]: ['bacon', 'mymeal', 'randint', 'spam', 'time']
```

  - **%reset** removes all variables.

```
In [17]: reset
```

```
Once deleted, variables cannot be recovered.
Proceed (y/[n])? y
```

```
In [18]: who
```

```
Interactive namespace is empty
```

Table 5.2. iPython Cell Magic Commands

Command	Description
`%%bash`	Equivalent to `%%script bash code`
`%%capture`	Runs the cell and captures the output.
`%%html`	Treats the cell as a block of html.
`%%javascript`	Treats the cell as a block of javascript.
`%%latex`	Treats the cell as a block of LaTeX.
`%%perl`	Equivalent to `%%script perl code`
`%%pypy`	Equivalent to `%%script pypy code`
`%%python2`	Equivalent to `%%script python2 code`
`%%python3`	Equivalent to `%%script python3 code`
`%%ruby`	Equivalent to `%%script ruby code`
`%%script`	Usage: `%%script language code`
`%%sh`	Equivalent to `%%script sh code`
`%%svg`	Renders the sell as an SVG (scalable vector graphics) literal.
`%%writefile`	Writes the contents of the of the cell to a file.

User Profile[9]

Configurable options, including anything that you want to have executed every time you start up iPython, are stored in the user profile.[10] To create your initial user profile go to the command shell or terminal (see page 22) and enter

```
$ ipython profile create
```

Then type the command

```
$ ipython locate
```

This will return a string listing the name of the folder that contains the profile. Most likely it will end with the name **.ipython**. The folder called "**.ipython**" contains the configuration files. Chances are that this is a hidden folder on your operating system. What this means is that if you go to the folder that contains "**.ipython**" by clicking on its folder icon in your desktop, the folder for "**.ipython**" won't be visible, unless you go to the trouble of making hidden folders visible.

Inside the "**.ipython**" folder look for the folder **profile_default**. Your default profile files are inside that folder. You can change your configuration by editing these files.

Suppose for example, that you want to change the default text editor to **geany**. Edit the file **ipython_config** in this folder. Look for the line with the variable

```
c.TerminalInteractiveShell.editor
```

and make sure that it is not commented (does not have a crosshatch character in front of it) and edit it to look like

```
c.TerminalInteractiveShell.editor = 'geany'
```

Then save your edits and exit the editor. The next time you start iPython and use the magic **%edit** command, it will run **geany** instead of **vm** (assuming that you have installed **geany** on your computer!).

[9] This section can be omitted by beginners.

[10] Full details on configuration and customization of iPython on given at http://ipython.readthedocs.io/en/stable/config/index.html

Exercises

1. Open the iPython shell on your computer. Try evaluating a few expressions in calculator more. Explain how it differs from the traditional Python shell.
2. Repeat the implementation of the Babylonian algorithm for $\sqrt{37}$, this time using the iPython shell.
3. Write a "Hello World" program using the iPython shell.
4. Do each of the following using magic commands:
   (a) Determine your current directory.
   (b) Change to a different directory.
   (c) Generate all the variables in your workspace.
   (d) Create a new list **u** that contains the values of all the variables in the current workspace.
   (e) Remove a variable from the workspace.
   (f) Verify that the variable has been removed.
   (g) Create a variable named **ls** by assigning it a value. Then make a list of all the folders in the current directory.
   (h) Use **timeit** to determine how fast your computer is.
5. Find your user profile file and print it out.

Chapter 6

Jupyter Notebooks

Jupyter notebooks arose from the iPython project; at one point some years ago, when all you could do in them was python, they were called iPython notebooks. Originally, the name jupyter was built from the names of three popular computer languages: JUlia, PYThon, and R (my capitalization). Nowadays there are something like forty or fifty different computer languages for which *kernels* have been written for jupyter (the correct capitalization is lower case). Each one of these kernels allow you to use a different computer language in a jupyter notebook. Besides **`python`**, **`R`**, and **`Julia`**, some of the other languages for which kernels have been written include (in no particular order): **`APL`**, **`Spark`**, **`Scala`**, **`Fortran`**, **`Ruby`**, **`C`**, **`C#`**, **`C++`**, **`F#`**, **`Go`**, **`Torch`**, **`Erlang`**, **`Forth`**, **`Perl`**, **`PHP`**, **`Octave Scilab`**, **`bash`**, **`Clojure`**, **`bash`**, **`Windows powershell`**, **`Hy`**, **`Mathics`**, **`Mathematica`**, **`Matlab`**, **`Lua`**, **`Scheme`**, **`IDL`**, **`Mochi`**, **`Brainfuck`**, **`Q`**, **`Jython`**, and **`Java`**.[1] The initial goal of these notebooks was to emulate concepts found in mathematical software such as **`Mathematica`**[2] and Sagemath.[3]

The key to jupyter notebooks is the **jupyter notebook interface**. The notebook interface uses a web browser, and when you edit, save, view, or run notebooks on your computer, you will use this jupyter notebook interface in your default web browser. It does not matter which web browser you use. In fact, jupyter will automatically figure out which browser you have selected on your operating system as your default browser and will always use that browser (unless you change some advanced settings).

Some students initially are confused by this: even though you are using a web browser, **you are not running jupyter on the internet**. Everything is running locally on your own computer, **it just looks like you are on the internet**. Jupyter is running software in the background that tells your computer to run what looks like a local server that only you and nobody else can see.

Furthermore, you can still use the internet and look up web pages while you are using jupyter, by opening new browser windows. What you do in jupyter will not affect your ability to use the internet. Just like Vegas, what happens in jupyter, stays in jupyter.[4]

The reason for using notebooks is this: notebooks display a work stream of text, graphics, code, and output in a single file. This file can be saved and later displayed or exchanged with another user to show how you performed a sequence of tasks. This is generally easier than giving somebody a collection of python files and instructions that they have to follow in a particular order. In a certain sense it resembles a lab notebook.

[1] A repository of kernels is available at https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

[2] See https://www.wolfram.com/mathematica/.

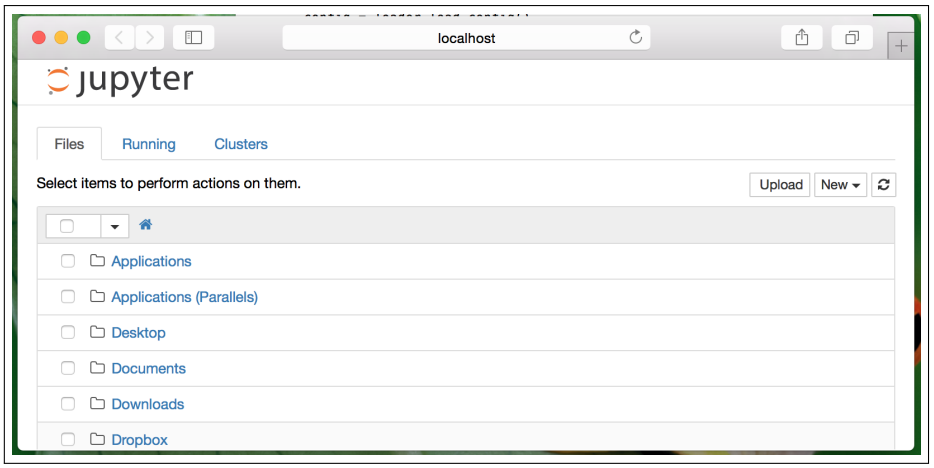
[3] See http://www.sagemath.org/.

[4] Of course, it is possible to access the internet from within jupyter or python programs, but that is a more advanced coding technique.

However, unlike a lab notebook you can edit out all of your drafts and mistakes and give a single, perfectly working notebook to another user. So the analogy is more to a laboratory instruction manual or protocol with examples than a notebook.

One caution about notebook files. They are coded in a special computer format called **JSON**. This format is never intended to be read by humans – it is designed to be written and read by other programs. The program with these **JSON** files is that if you double-click on a notebook icon on your computer and open it in a text editor you will see **JSON** code and not python code. You should not attempt to edit or otherwise modify this **JSON** code. You should only read and write this file through the jupyter notebook interface.

Figure 6.1: The jupyter iterface on a mac. Open this interface by typing **jupyter notebook** in a terminal window, such as **cmd.exe**. When you first open the interfae all you see is a listing of accessible files and folders. Notebooks will end with the letters **.ipynb**.



To open the notebook interface you need to go to the command line (the terminal program such as **cmd.exe**, **terminal** or **powershell**) and type in the line[5]

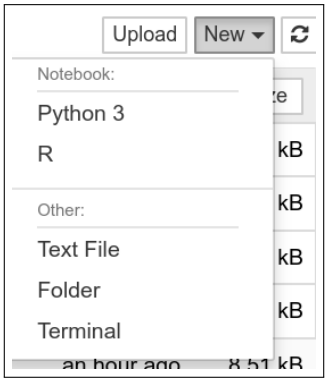
```
$ jupyter notebook
```

This command will start a local web server running on your computer that can read and write iPython notebooks. Don't worry – you are not setting up a web site. This is a special type of web server that you can only access from your computer. Just ignore the stuff that scrolls down the terminal. At the same time, a new browser window will open in whatever is the default browser in your operating system. An example of what you will see is shown in figure 6.1.

To create a new python 3 notebook, pull down the menu New > python 3 (fig. 6.2). Your

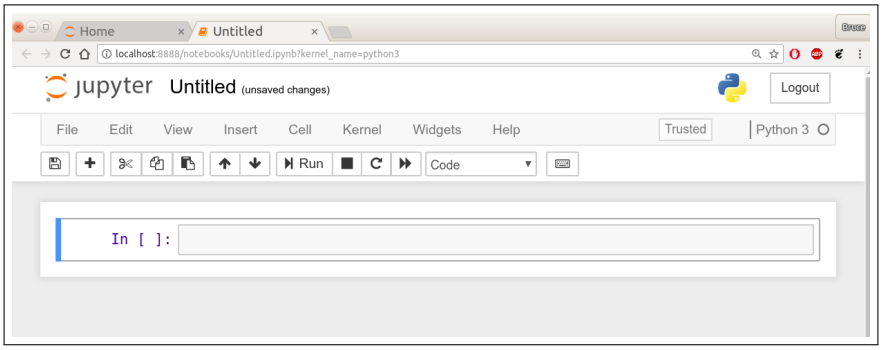
[5]Some python distributions, such as Enthought, will come with launchers. You avoid these interfaces until they know what they are doing, because they add in shortcuts that break Python standards, such as including all of **pylab** without telling you. Notebooks created in this manner are not compatible with standard Python, and should be used with caution. Don't be surprised if you create a notebook in Canopy and turn it in only to get a failing grade because it crashed when your professor rand the program, because you didn't know you had to include some library that Canopy automatically included without telling you!

Figure 6.2: Creating a new notebook from the menu.



new notebook appears in a tab labeled Untitled (fig. 6.3).

Figure 6.3: A new empty notebook.



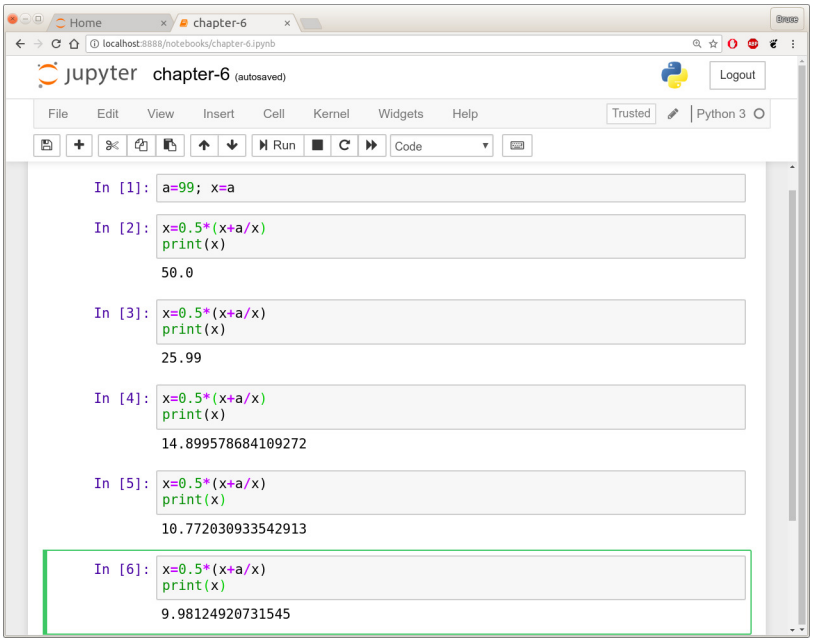
Notebooks are organized in pieces that are called cells; these are like paragraphs or chapters in a book. Each cell can contain whatever you want: a function, a piece of code, a picture, an essay, some documentation, or some program output. The empty notebook we have just created has a single empty **cell** in it labeled(for now) `In [ ]:`. The word `In` means it is an input cell, and the fact that nothing is written between the square brackets means that we have not executed the cell.

The numbers inside the brackets are filled in sequentially in the order in which they are are run. So if you go back to an earlier cell in your notebook, modify the value, and re-run it, the output of that cell will show the output with the new value. If subsequent cells in the notebook depend on this value, but are not re-run, any output they have calculated will remain unchanged. This can lead to some confusion in reading notebooks if you scroll back-and-forth making changes.

A cell is executed by putting the cursor anywhere in the cell (with the mouse) and then typing shift+enter (i.e., by hitting the shift and enter keys simultaneously). Hitting the enter alone will just add a new line to the cell. Thus it is possible to execute cells out of order in a notebook.

Suppose we enter the code `x=99` and then press shift+enter. Nothing much happens

Figure 6.4: Several calculations of the Babylonian Algorithm in a jupyter notebook.



except that the number 1 appears between the brackets. But then this is what we would also expect in the Python shell. If we now try to calculate one iteration of the Babylonian algorithm and print out the result, we see something different. It creates output cells (figure 6.4).

A list of keystroke commands is available by hitting escape + h.

To add new cells either above or below an existing cell in a notebook, hit the escape + a or escape + b, respectively. Think of "above" and "below" (not "before" and "after", which will just confuse you).

To toggle line numbers on or off in a single cell, use escape + l.

Notebooks are particularly useful for combining both text and graphics. Instead of using a **show()** command (chapter 23) all graphics output can be redirected to the notebook and the corresponding **show()** call omitted if you include the command

```
In [1]: %matplotlib inline
```

before your first graphics command.

Additional cells can be added for documentation. If these cells are labeled as markdown using the drop-down menu in the center of the toolbar that is normally listed as code, then formatting commands can be added using both html and latex.

Complete documentation of the iPython notebook is given on the official website at http://ipython.org/documentation.html.

Exercises

1. Create a new jupyter notebook on your system. Perform each of the following.
   (a) Type an expression, like **x=3**, into the first cell, ad evaluate it.
   (b) Type two lines of expressions that depend on x into the second cell, like

```
In [2]: y=x+3
        print(x+y)
```

   and evaluation them.
   (c) Place your cursor in the first cell and use esc + b to create a new cell below the current cell. Put an expression there that changes the value of **x**. The select cell > run all from the menus. What happens?
   (d) Print a list of all the magic command.
   (e) Display a list of keyboard shortcuts.
   (f) Rename your notebook.
   (g) Quit jupyter and exit all browsers.
   (h) Restart **jupyter notebook** from the command shell.
   (i) Open the notebook you just saved. Execute it by selecting cell > run all.
2. In a notebook create a cell that defines a function **f(x,a)** to do one update of the Babylonian algorithm for $\sqrt{a}$.
3. Using the same notebook that you used in 2, write a program in a different cell, that calculates $\sqrt{42}$. It should call the function you wrote in the earlier exercise, which is already in the notebook.
4. In the same notebook that you used in 2, go to the top of your notebook and insert a new cell. Write the line **import math**, and then execute the cell. Scroll back down to the bottom of the notebook, and calculate $\sqrt{42}$ using **math.sqrt(42)**. Compare your answer with the number you calculated using the Babylonian algorithm. Create a new Markdown cell and type in your observations.
5. Any cell can be converted to markdown by placing the cursor inside of it and then clicking on the menu that normally says code and changing it to say markdown. You can also place the cursor in the cell and type ctrl + m followed by m. Insert various markdown cells in a notebook and experiment with different formats.[6] Verify that you can embed each of the following in a markdown cell:
   (a) Bulleted lists and sublists.
   (b) Itemized (numbered) and sublists.
   (c) Headings and subheadings.
   (d) Latex expressions.
   (e) HTML expressions.
   (f) Links to files or URLS.
   (g) Plain old regular text.
6. Read the official tutorial.[7]
7. Familiarize yourself with the official online documentation.[8]

[6] The syntax for markdown is described at http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Working%20With%20Markdown%20Cells.html.
[7] This can be found at http://ipython.readthedocs.io/en/stable/interactive/tutorial.html?highlight=tutorial.
[8] This can be found at http://ipython.readthedocs.io/en/stable/overview.html.

Chapter 7

Numbers in Computers

Binary Numbers

The information in a computer is stored in binary (base 2). This is because of the hardware design of all computers used today, which are based on switching circuits. At the very heart of their circuits are simple components called **switches** which can be in either of two states: **on** or **off**. These two states are used to represent the numbers zero and one. The term **bit** is a shortened form of the oxymoronic term **binary digit**.[1] Consequently, modern computers are called **digital computers** (again oxymoronically, since they are really binary).

Consider the base ten number 117.[2] To understand how 117_{10} is stored in the computer, we first recall from elementary school arithmetic that

$$117_{10} = (1 \times 10^2) + (1 \times 10^1) + (7 \times 10^0) \tag{7.1}$$

To represent the same number in base two, we need to expand it in **powers of 2** instead of **powers of 10**. Since

$$117 = 64 + 32 + 16 + 4 + 1 \tag{7.2}$$

$$= (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \tag{7.3}$$

we know that there must be 1's in the 1, 4, 16, 32, and 64 places, and zeros in the 2's and 8's places. So we have

$$117_{10} = 0111\ 0101_2 \tag{7.4}$$

When typesetting binary numbers, a space is typically left every fourth bit. This space is analogous to the comma[3] that is normally used every third place when typesetting decimals. The four-bit grouping is useful because it makes conversion to **hexadecimal** (base 16) much easier. In addition to the digits 0 through 9, hexadecimal also uses the letters A through E to represent the numbers 10 through 15:

[1] The term **digit** refers exclusively to base 10, because we have ten fingers (digits).

[2] When there is any confusion about the base, we use a subscript next to the number to represent the base. Instead of writing "117 in base 10" we write 117_{10}.

[3] Commas are used every third digit, and a period is used for a decimal point, in North America. In much of the rest of the world, however, the period and the comma are interchanged, so that 5,678.45 becomes 5.678,45.

$0 \cdots 9$ represent $0 \cdots 9$
A represents 10
B represents 11
C represents 12
D represents 13
E represents 14
F represents 15

Since

$$117_{10} = (7 \times 16^1) + (5 \times 16^0) \tag{7.5}$$

we can write

$$117_{10} = 75_{16} \tag{7.6}$$

We can also convert 4 bit grouping individually to hexadecimal:

$$117_{10} = \underbrace{0111}_{7}\ \underbrace{0101}_{5} = 75_{16} \tag{7.7}$$

A similar direct conversion to **octal** (base 8) could be made by collecting the bits in groups of three instead of groups of four.

Nibbles, Bits, and Bytes

The basic hardware unit of memory is a **byte**, which represents eight bits. It is convenient, sometimes, to think of a byte as a place with eight "slots," each of which can contain a one or a zero. For example, we might write 117_{10} as

$$117_{10} = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\ \ \boxed{0}\boxed{1}\boxed{0}\boxed{1} \tag{7.8}$$

Each block of four bits is called a **nibble**.

Since a nibble has four slots, it can represent any of $2^4 = 16$ different numbers.

A byte has 8 bits or can represent any of $2^8 = 256$ different numbers.

A **word** consists of 4 bytes or 32 bits, and can represent up to $2^{32} = 4,294,967,296$ different numbers. The four-gigabyte limit on old computers was a result of a computer addressing scheme that counted memory locations using a single 32 bit word.

A **double-word** consists of 2 words, or 64 bits, and can represent up to $2^{64} = 18,446,774,073,709,551,616$ different numbers. Most computers sold today use 64-bit memory.

Because $2^{10} = 1024 \approx 1000$ the word "kilo" became popular to represent chunks of 1024 bytes in the early days of computing, in analogy with its use in the metric system to represent chunks of 1000 litres or 1000 metres. This led to a relatively small error (2.4%) that most people did not care about. But this error is compounded when larger amounts of memory are computed. Consequently, to represent memory correctly, the correct nomenclature (rarely used) is that $2^{10} = 1024$ bits = 1 **Kibibit**, while 1000 bits = 1 **kilobit**. Unfortunately these terms are almost always used incorrectly.

Value	Letter	Name	Value	Letter	Name
1000	k	kilo	1024	Ki	kibi
1000^2	M	mega	1024^2	Mi	mebi
1000^3	G	giga	1024^3	Gi	gibi
1000^4	T	tera	1024^4	Ti	tebi
1000^5	P	peta	1024^5	Pi	pebi
1000^6	E	exa	1024^6	Ei	exbi
1000^7	Z	zetta	1024^7	Zi	zebi
1000^8	Y	yotta	1024^8	Yi	yobi

Example 7.1. Misleading Computer Hardware Labels. You purchase a new hard disk for your computer. The label on the box says 1.5 terabytes, and one terabyte $= 1000^4$ bytes. You plug it into your computer and it says it has a capacity of only 1.36 TB. What happened to the other 140 MB?

The computer is calculating the memory in tebabytes:

$$1.5 \text{ terabytes } \times \frac{(1000)^4 \text{ bytes/terabyte}}{(1024)^4 \text{ bytes/tebabyte}} = 1.36424 \text{ tebabytes} \tag{7.9}$$

All operating systems display memory in units of 1024 Bytes. The manufacturer is being honest when it says that it is selling you 1.5 terabytes - because a terabyte is 10^{12} bytes. Your computer is also being honest, because it is not displaying the information in terabyptes but in tebibytes, and a tebibyte is $1024^4 = 1.024^4 \times 10^{12}$ bytes. Thus 1.5TB = 1.36TiB. Manufacturers are always going to label things in the largest possible number, to make it look like you are getting the most for your money. To the computer, however, that 1.5 has no significance, and it only measures things in units of 1024 bytes.

Integer Representation

Computer words are just strings of ones and zeros, so there is no way intrinsically to represent a negative number. Suppose, for example, that we have a computer with 4-bit words. Every word in our hypothetical computer can take on up to 16 different values. If we write these as

$$0000_2 = 0_{10},\ 0001_2 = 1_{10},\ 0010_2 = 2_{10}, \ldots, 1110_2 = 14_{10},\ 1111_2 = 15_{10} \tag{7.10}$$

then we are only representing the non-negative numbers $0 \leq n \leq 15$. What we would really like is to have around half of the numbers represent negative numbers, and around half represent positive numbers.

There are several common ways that the interpretation of the number stored in memory can be changed so that we treat half of the numbers as negative. The actual representation varies from computer to computer.

Use a Sign Bit. We can use let the first bit represent the sign: If the sign bit is 0, treat the rest of the number as positive; if the sign bit is 1, treat the rest of the number as negative.

0000 = 0	1000 = -0
0001 = 1	1001 = -1
0010 = 2	1010 = -2
0011 = 3	1011 = -3
0100 = 4	1100 = -4
0101 = 5	1101 = -5
0110 = 6	1110 = -6
0111 = 7	1111 = -7

As we can see there are two different representations for zero, but we can represent all integers from $-7 \leq n \leq 7$ with this scheme.

One's complement. In one's complement notation, we use a sign bit as well, but if the sign bit is 1, we first flip all the bits and then treat the number as negative. If the sign bit is zero we read the number as is and treat it as positive.

0000 = 0	$1000 \rightarrow 0111 = -7$
0001 = 1	$1001 \rightarrow 0110 = -6$
0010 = 2	$1010 \rightarrow 0101 = -5$
0011 = 3	$1011 \rightarrow 0100 = -4$
0100 = 4	$1100 \rightarrow 0011 = -3$
0101 = 5	$1101 \rightarrow 0010 = -2$
0110 = 6	$1110 \rightarrow 0001 = -1$
0111 = 7	$1111 \rightarrow 0000 = -0$

The consequence is the same as before; we still have two representations for zero, and range of -7 to 7.

Excess-p notation. Let $p = 2^{n-1} - 1$ where n is the number of bits. For our four-bit machine, we have $p = 7$. We assume that the number represented by the computer is just off by 7.

0000 = 0-7 = -7	1000 = 8-7=1
0001 = 1-7 = -6	1001 = 9-7=2
0010 = 2-7=-5	1010 = 10-7=3
0011 = 3-7=-4	1011 = 11-7=4
0100 = 4-7=-3	1100 = 12-7 = 5
0101 = 5-7=-2	1101 = 13-7=6
0110 = 6-7=-1	1110 = 14-7=7
0111 = 7-7=0	1111 = 15-7=8

This time we only have one representation of zero, and we have gained an extra integer: $-7 \leq n \leq 8$.

Two's complement notation. In this representation, positive numbers are represented normally and negative numbers are represented by flipping their bits and adding 1. This if the first bit is a 1 we automatically know the number is negative.

$$
\begin{array}{ll}
0000 = 0 & 0111 + 1 \rightarrow 1000 = \text{-}8 \\
0001 = 1 & 1110 + 1 \rightarrow 1111 = \text{-}1 \\
0010 = 2 & 1101 + 1 \rightarrow 1110 = \text{-}2 \\
0011 = 3 & 1100 + 1 \rightarrow 1101 = \text{-}3 \\
0100 = 4 & 1011 + 1 \rightarrow 1100 = \text{-}4 \\
0101 = 5 & 1010 + 1 \rightarrow 1011 = \text{-}5 \\
0110 = 6 & 1001 + 1 \rightarrow 1010 = \text{-}6 \\
0111 = 7 & 1000 + 1 \rightarrow 1001 = \text{-}7
\end{array}
$$

Two' complement has the advantage that mathematical operations work normally – no special hardware is needed. The math just comes out right. Furthermore, there is only a single representation for zero; there is no "negative zero," like there is in the one's complement notation. Nearly all computers implement the two's complement method in their hardware, although some early computers used the other techniques.

Numbers with Decimal Points

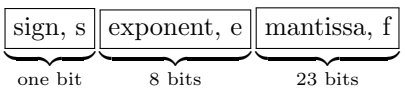
There is no decimal point in a computer word so we cannot represent anything except for integers directly. Instead, we approximate real numbers by their **floating point representation**

$$x = (\pm 1) \times (m) \times 10^E \tag{7.11}$$

where E is an integer **exponent**, m is an integer **mantissa**, and one bit is reserved for the **sign**.

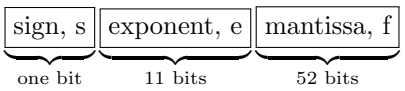
There are three standard ways of doing this, depending upon whether one, two or four words of 32 bit memory are used. Understanding these implementations will help you to understand why computations work the way they do.

In the **IEEE 32 bit standard**[4] a single 32 bit word of memory is used to represent a floating point number.



Since there are $2^8 = 256$ possible exponents, there is a range of approximately $10^{\pm 38}$ values;[5] and since there are 23 bits in the mantissa, there are $2^{23} = 8,388,608$ possible mantissas, which gives approximately 7 significant figures. This is called **single precision** in some languages, but is not used in Python.

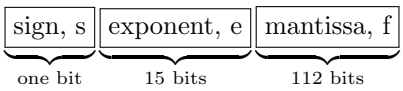
In the **IEEE 64 bit standard** two 32 bit words of memory (on older computers) or one 64 bit word of memory is used to represent a floating point number. This is the representation used by Python.



[4] IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754, revised 2008.
[5] To get this solve $10^n \approx 2^{127}$ for n - reserving one exponent bit for the sign.

There are 11 bits in the exponent, so that $2^{11} = 2048$ possible exponents. This gives a range of approximately $10^{\pm 308}$. The 52 bit mantissa has $2^{52} \approx 7.2 \times 10^{16}$ possible values, giving approximately 15 significant figures. This representation is called **double** or **double precision** in some languages.

In the **IEEE 128 bit standard** four 32 bit words, or two 64 bit words are used for a total of 128 bits.



There are $2^{15} = 32,768$ possible exponents, giving a range of approximately $10^{\pm 4931}$. There are $2^{112} \approx 5.2 \times 10^{33}$ possible mantissas, for approximately 33 significant figures. This representation is called **quadruple precision** in Fortran or **long double** in some C/C++ compilers.

Exercises

1. Convert from decimal to binary:
   (a) 100 (b) 142 (c) 1/3 (d) 0.1
2. Convert from octal to binary:
   (a) 73 (b) 462 (c) 7632 (d) 101.101
3. Convert from binary numbers to decimal, octal and hexadecimal:
   (a) 110101 (b) 1101.111
   (c) 110110111 (d) 1.101010101
4. Convert from hexadecimal to decimal:
   (a) 10 (b) A1 (c) FF1A (d) 1A
5. Referring ahead to chapter 10, write python representations of:
   (a) Each octal integer in exercise 2.
   (b) Each binary integer in exercise 3.
   (c) Each of the hexadecimal numbers in exercise 4.

Chapter 8

When Numbers Fail

In Python the standard floating point representation gives us about 15 significant figures. Why do we need so much precision in floating point numbers?

There are two basic problems. First, not all decimal numbers can be represented precisely in binary. For example, 1/10 is a repeating decimal in base 2, and whenever we truncate, we get an error:

$$0.110 = 0.0001\,1001\,1001\,\overline{1001}_2 \tag{8.1}$$

Second, most numerical computations will include a large number (millions or billions or more) of repeated calculations, and even small errors can build up over time.

Roundoff Error

To understand roundoff error, we will consider a computer that stores its numbers in base 10 rather than base 2, because as human beings we can understand base 10 more easily. Let's use the Babylonian algorithm to calculate $\sqrt{a}$ with 3-digit truncation. This means we truncate every calculation to 3 digits. In the algorithm, we start with some initial guess x_0 and then iterate on

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{8.2}$$

until $|x_{n+1} - x_n| < \epsilon$ for a desired tolerance ϵ. For example, here are some possible iterations for several values of a and different starting values.

a	4.00	1.00	0.800	0.950	0.999
x_0	2.50	0.500	0.900	0.970	0.995
x_1	2.05	1.25	0.890	0.970	0.995
x_2	2.00	1.02	0.890	0.970	0.995
x_3	2.00	1.00	0.890	0.970	0.995
exact	2.00	1.00	0.894	0.970	0.999

In each case, if there are extra digits, they just fall off the right end of the storage space for the number and are lost. This is equivalent to dropping (or truncating) the last digits.

In the third column,

$$x_1 = (0.900 + .800/.900)/2 = (.900 + .888)/2 = 1.78/2 = 0.890 \tag{8.3}$$

In the fourth column,

$$x_1 = (0.970 + .950/.970)/2 = (0.970 + .979)/2 = 1.94/2 = 0.970 \tag{8.4}$$

In the last column the error is even worse, as it moves *away* from the correct answer.

$$x_1 = (0.995 + 0.999/.995)/2 = (0.995 + 1.00)/2 = 1.99/2 = .995 \tag{8.5}$$

The answer is worse than the first guess, even though we know the algorithm is *mathematically* guaranteed to converge to the correct answer.

The problem in all of these cases is that we are calculating the average of two numbers, and the formula $(a + b)/2$ will not always work. A worst-case scenario would be something like the average of 0.501 and 0.503:

$$\frac{0.501 + 0.503}{2} = \frac{1.00}{2} = 0.5 \tag{8.6}$$

which is not even between 0.501 and 0.503. This is because $0.501 + 0.503 \neq 1.004$, because we are only using three digits. The fourth digit is lost. So we end up with an average that is not even between the two original numbers. A better formula for the average would be

$$\text{average} = a + \frac{b - a}{2} \tag{8.7}$$

Then

$$.501 + \frac{.503 - .501}{2} = .501 + \frac{.002}{2} = .501 + .001 = .502 \tag{8.8}$$

The worst case happens when the average falls on one endpoint, e.g.,

$$.501 + \frac{.503 - .502}{2} = .501 + \frac{.001}{2} = .501 + .000 = .501 \tag{8.9}$$

At least now the average is between the two end points (inclusively).

Numerical Errors: Dhahran, 1991

From the New York Times:

> DHAHRAN, Saudi Arabia, Tuesday, Feb. 26, 1991 In the most devastating Iraqi stroke of the Persia Gulf war, an Iraqi missile demolished a barracks housing more than 100 American troops on Monday night, killing 27 and wounding 98, the American military command in Riyadh said early today.

Three months later, on May 29, the same newspaper printed this:

> The Iraqi missile that slammed into an American military barracks in Saudi Arabia during the Persian Gulf war, killing 28 people, penetrated air defenses because a computer failure shut down the American missile system designed to counter it, two Army investigations have concluded ... The radar system never saw the incoming missile, said Col. Bruce Garnett, who conducted one of the investigations...

What was behind this failure of the American PATRIOT[1] anti-missile system to shoot down the incoming Iraqi Scud missile? The Scud missile was travelling at Mach 5 (five times the speed of sound), approximately 6000 km/hour or 1700 meters per second. The PATRIOT uses a radar system to track and shoot down precisely this type of missile.

The software in the PATRIOT stored time as a 24-bit number in units of 1/10 of a second. Since 1/10 cannot be represented exactly in based 2 it was truncated as

$$0.1_{10} \approx 0.0001\,1001\,1001\,1001\,1001\,100_2 \tag{8.10}$$

or

$$0.1 \approx 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} \tag{8.11}$$

$$= \frac{209,715}{2,097,152} \tag{8.12}$$

Therefore the error that was made with each tick of the clock was

$$\epsilon = \frac{1}{10} - \frac{209,715}{2,097,152} = \frac{1}{10,485,760} \approx 9.54 \times 10^{-8} \tag{8.13}$$

every 0.1 second:

- After one second, the accumulated error in time is 9.54×10^{-7} seconds
- After one minute, the accumulated error is 0.0000572205 seconds
- After one hour, the accumulated error is 0.00343323 seconds
- After one day, the accumulated error is 0.0823975 seconds
- After 100 hours, the accumulated error is 0.343323 seconds

At the time of the attack, the computer had been running for approximately 100 hours without being reset. So there was an accumulated error of 0.3433 seconds. Since the Scud was travelling at approximately 1.7 km/second, the radar was expecting the missile to be approximately 580 meters away from where it actually was. Since it wasn't where it should have been, it was not classified as a threat.

The programmers knew about the accumulated error problem but this information had not been communicated to the operators of the system in combat for undetermined reasons (chain of command, security, who knows?) A fix for the software was in transit at the time of the attack. Had the attack occurred a week later the PATRIOT would have intercepted the incoming missile.

Atlanta Airport, 2009

From the Los Angeles Times, 20 Nov 2009:

> **U.S. flight delays hit Atlanta airport especially hard: The FAA blames a computer glitch in Salt Lake City**. Hundreds of flights

[1] *Phased Array Tracking Radar to Intercept Of Target*, aka *Protection Against Threats, Real, Imagined, or Theorized.*

> around the country were canceled or delayed Thursday after a communications failure at a Federal Aviation Administration computer center, leaving passengers scrambling to revise travel plans. The glitch, which occurred about 5 a.m. Eastern time, prevented airlines from electronically entering their flight plans into an FAA computer in Salt Lake City that air traffic controllers nationwide rely on.

But what is a **computer glitch**?

> **There is no such thing as a computer glitch**. The word glitch has no meaning. **Either someone wrote an incorrect program, or someone did not run the program properly**. Someone did not do his or her job correctly. The only reason we use a word like glitch is that we then do not have to hold anyone accountable. (Tom Impelluso, Los Angeles Times, 24 Nov. 2009)

A computer glitch is a problem that has not been solved. It is not a problem with the computer. It is because the computer was incorrectly used. In the case of the PATRIOT missiles, the error was leaving on the system for too long but not telling the users that was not a good thing. The error in the FAA software was not identified.

USS Yorktown, 1997, and Windows NT: Dead in the Water

The USS Yorktown was a Aegis missile cruiser in the US Navy launched in 1984. In the early 1990's it was retro-fitted with a new Engineering Control and Integrated Bridge System built by Litton Industries in Woodland Hills, California, at a cost of nearly $140 Million. In these "smart ships" all on-board systems were controlled by applications running under Windows NT. However, when bad data was fed into one of the computers during maneuvers off Cape Charles, VA, on 21 Sept., 1997, a division by zero error occurred in the system software. This error caused the entire on-board computer network to crash; the ship's crew were unable to bring systems back up again. This made the ship entirely "dead in the water" for about two and half hours (quote from Commander John Singley of the Atlantic Fleet Surface Force, archived in *Wired*). The ship had to be towed to port. The Yorktown was decommissioned in 2004.

F-22 Raptors, 2007: The International Date What?

In the first deployment of twelve new F-22 Raptors to Okinawa, while en-route from Hawaii in 2007, as they crossed the international dateline all their systems failed because of a software error that assumed this event would never occur. The pilots managed to navigate the aircraft back to Hawaii by manually following a tanker.

Google, 2009: The entire internet is spam

On Jan 30, 2009, a programmer at Google accidentally entered "/" as the address of a malware site. Since this designation is used to represent the top-level directory (root) of a computer, the effect was to designate the entire internet as malware. For all practical purposes, Google had shut down internet searches for 40 minutes.

Therac-25, 1987: Killing Cancer Patients with x-Rays

The Therac-25 was used to treat cancer patients in the 1980's with radiation therapy, but the error messages were ambiguous and confusing to operators. Under certain conditions the radiation beam could become thousands of times stronger than expected. Rather than shutting off the machine when this happened, the failsafe software merely printed a message "MALFUNCTION n," where n was a number between 1 an 64. Users did what they do best, which was to ignore the error messages, and several patients were killed.

Numerical Error

Even if your program is completely perfect and error free, there could still be some numerical inaccuracy to it. These **inherent numerical errors** in computations have two general sources. First, there may be some inaccuracies in the input data. This error is then propagated forward, and compounded, by successive computations. This type of error is called **data error**. Data error is causes by errors that are already present before a computation begins. Sources include:

1. **Measurement errors**: The number supplied to the program could be wrong.
2. **Previous computations**: Successive computations depend on earlier computations. If the result of one computation that has an error in it, and is used as the input to another computation, this causes a data error.
3. **Modeling errors**: The theory behind the implementation could be approximate. For example, one might model a gravity force by $\mathbf{F} = -m\mathbf{g}$, an approximation that is only valid near the surface of the earth, instead of $\mathbf{F} = -GMm/r^2$.

A second type of errors are those introduced by the computation itself. These **computational errors** are usually due to the limitations of the actual hardware or software implementation. Computational errors may be caused by

1. **Roundoff errors**: Errors due to the fact that computers use a finite number of digits to represent numbers.
2. **Truncation errors**: Errors due to the truncation of an infinite process, such as calculating only a finite number of terms in a Taylor Series approximation.

Of course, there can always be a bug in your program that will lead to a numerical error; for example you might transcribe a formula incorrectly when you implement it, or use a particular programming construct incorrectly.

Lets assume your program is perfect and bug free, and try to quantify the inherent error.

Definition 8.1. Absolute Error

The absolute error in a numerical approximation is

$$\textbf{absolute error} = |\text{approximate value - true value}| \tag{8.14}$$

Definition 8.2. Relative Error

The relative error in a numerical approximation is

$$\textbf{relative error} = \frac{\text{absolute error}}{\text{true value}} \tag{8.15}$$

This gives us the useful result

$$\text{approximate value} = (\text{true value}) \times (1 \pm \text{relative error}) \tag{8.16}$$

We will sometimes use the term **unit in the last place** or **ulp** to represent the value of a 1 when placed in the rightmost digit of a numerical representation of a number.

Definition 8.3. ULP

One ULP - Unit in the Last Place – is a 1 in the rightmost digit of a number.

Example 8.1. Suppose that $e = 2.718281828459045...$ is represented in a computer with a 23 bit mantissa. Find the ULP.

$$\text{True Value: } e = 2.718281820459045...$$
$$\text{Approximate Value: } \hat{e} = 2.71828135 = 10.\underbrace{1011\cdots101}_{21 \text{ bits}}{}_2$$
$$1 \text{ ULP} = 2^{-21} \approx 4.768 \times 10^{-7}$$

Here is a program to find the ULP in Python.

```
1 ulp = 1.0
2 iterations = 0
3 while ((1+ulp) - 1) > 0:
4    ulp = ulp/2
5    iterations += 1
6 print(ulp)
7 print(iterations-1)
```

This loop will terminate after 55 iterations with an ulp of approximately 1.1×10^{-16}. When that number is added to 1, the result is 1, and so `((1+ulp)-1)` gives zero and the loop terminates.

If we had written line 3 in the above loop as `while ulp >0:` the loop would always return zero, because the number `ulp` will continue to divide by 2 until the program reaches 4.9×10^{-324} after 1075 iterations. When this is divided by 2, it falls beneath the lowest precision available and returns zero.

Definition 8.4. Decimal Places of Accuracy

$$\textbf{decimal places of accuracy} = \lfloor -\log_{10}(\text{absolute error}) \rfloor \tag{8.17}$$

The notation $\lfloor x \rfloor$ means the greatest integer less than or equal to x (the floor function).

The decimal places of accuracy gives approximately the number of digits that are accurately represented to the right of decimal point. In example 8.1 we had 1 ulp $= 4.768 \times 10^{-7}$, so that an error of 1 ulp represents $\lfloor (-\log 4.768 \times 10^{-7}) \rfloor = \lfloor 6.322 \rfloor = 6$ decimal places of accuracy.

We are sometimes only interested the total number of correct digits, not just the number of digits to the right of decimal point that are correct. This is derived from the relative error.

Definition 8.5. Digits of Accuracy

$$\textbf{digits of accuracy} = \lfloor -\log_{10}(\text{relative error}) \rfloor \tag{8.18}$$

The digits of accuracy gives approximately the total number of digits of accuracy, starting from the first nonzero digit. So 3.124, 3124, and 0.003124 all have 4 digits of accuracy, whereas they have 3, 0, and 6 decimal places of accuracy, respectively. In example 8.1 we had a relative error of $r \approx (4.768 \times 10^{-7})/e \approx 1.75 \times 10^{-7}$. Since $\log(4.768 \times 10^{-7}) \approx -6.76$, there are 6 digits of accuracy to this estimate.

Once there is an error in a quantity, it will be propagated by the program. We can quantify this as the propagated data error.

Definition 8.6. Propagated Data Error

Let x be a true value of a quantity, $\tilde{x}$ be the same quantity with data error, and let $f(x)$ represent what we are trying to compute. Then

$$\textbf{propagated data error} = f(\tilde{a}) - f(a) \tag{8.19}$$

The propagated data error defined in this way has nothing to do with the computer implementation of f: it only depends on the true definition of f.

Example 8.2. Estimate the propagated data error due to calculating $\cos(\pi/3)$ using $\pi = 3.1416$ Since we have approximated π at the fourth decimal place, this is inherently inaccurate.

$$\text{propagated data error} = \cos(3.1416/3) - \cos(\pi/3) = -2.12 \times 10^{-6} \tag{8.20}$$

The computational error depends on the way in which we calculate f.

Definition 8.7. Computational Error

Let x be some input data, f describe a function we are computing, and $\hat{f}(\tilde{x})$ the computed value using an approximate value $\tilde{x}$ for x. Then

$$\textbf{computational error} = \hat{f}(\tilde{x}) - f(\tilde{x}) \tag{8.21}$$

Example 8.3. Estimate the computational error for $\cos(\pi/3)$ using a three-term Taylor approximation for $\cos x$ and $\pi \approx 3.1416$.

We have $\hat{f}(x) = \cos x \approx 1 - \dfrac{x^2}{2} + \dfrac{x^4}{24}$ and therefore the computational error is

$$\text{error} \approx 1 - \frac{(3.1416/3)^2}{2} + \frac{(3.1416/3)^4}{24} - \cos\frac{3.1416}{3} \tag{8.22a}$$

$$\approx 0.5017941058109612 - 0.4999978792725457 \approx 0.001796 \tag{8.22b}$$

Exercises

1. Suppose you have a computer that stores numbers with three digits of accuracy. Suppose it uses truncation error (lopping off the extra digits) when overflow occurs. How does it store (or calculate, and then store):
   (a) 5/9 (b) 5/7
   (c) (5/9)+(5/7) (d) (5/9)-(5/7)
2. Repeat the previous problem assuming that there is some kind of special hardware that performs round-off after each calculation.
3. Suppose you have a digit that stores numbers with 3 digits of accuracy. Let $x = 566$, $y = 568$, and $z = 999$. What would your computer calculate for the average of u and v, where $u = x/z$ and $v = y/z$, assuming it uses each of the following formulas:
   (a) average $= (u + v)/2$
   (b) average $= u + (v - u)/2$
4. Suppose you have a computer that does calculations to 4 significant figures. Calculate the first iterations to $\sqrt{1}$ in the Babylonian algorithm using (a) $x_0 = 0.999$; (b) $x_0 = 1.001$
5. In the Python shell type in the following expression:

   ```
   >>>  3.0/10-3*.1
   ```

   Observe and explain your results.
6. If a SCUD missile should have been within 50 meters of its predicted positions for it to be classified as a danger, what is the maximum amount of time the system could be left on without a reset?
7. Determine the ULP of your computer.
8. Suppose you were writing a program to calculate $\cos(x)$ using the Taylor series approximation. How many terms would you have to include to calculate the result to machine accuracy? Hint: this is an alternating series. Look up the theorems for convergence of an alternating series in your calculus book.
9. Consider the quadratic $x^2 + 500x + 1 = 0$.
   (a) Find the roots in a four-digit truncation system using the quadratic formula in the following form:
   $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
   (b) Repeat part (a), but this first rearrange the quadratic equation (analytically) by rationalizing the numerator, and use the resulting formula.
   (c) Compare your two sets of results. The actual roots are at $-1/(250 + \sqrt{62499}) \approx -.002$ and $-250 - \sqrt{62499} \approx -499.998$. Can you come up with a general rule about when to use the normal formula and when to rationalize the numerator?

Chapter 9

Big Oh

Algorithms (definition 3.1) describe how computers process information. They specifically state **what happens** at each step, but are **implementation independent**. To be useful an algorithm should be **correct**, **efficient**, and **implementable**.

One important measure of an algorithm's efficiency is the number of steps it takes to complete. By counting the number of **basic operations** performed by the algorithm, we can get a device-independent measure of how fast an algorithm works.

For non-mathematical algorithms, the number of operations is the number of lines of code executed. Thus if a **for** loop that contains a suite of 15 executable lines of code loops through 20 iterations, the number of basic operations is $20 \times 15 = 300$. If any of those lines of code were function calls (chapter 18), the weight of the line of code for each function would have to be replaced by the number of basic operations performed within the function.

For mathematical computations, basic operations are multiplications and additions. Consider matrix multiplication (definition B.9 or example 19.5). Let c_{ij} be the ij^{th} element of the product of two square $n \times n$ matrices $\mathbf{A}= [a_{ij}]$ and $\mathbf{B}=[b_{ij}]$. Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj} \tag{9.1}$$

Just to calculate c_{ij} there are n multiplications and $n-1$ additions, or $2n-1$ operations. Since we have to repeat this operation $n \times n = n^2$ times, the total number arithmetic operations are $n^2(2n-1) = 2n^3 - n^2$.

To get an idea of what this means for real calculations, we need to know the machine speed. An Intel Core i7 5960x Haswell chip (the state of the art desktop PC in 2015) runs at approximately 300,000 **MIPS**. A MIP is a million instructions per second, so the Haswell completes approximately 3×10^{11} instructions per second (IPS). The K-computer, with over 700,000 cores, and which was ranked the fastest computer in the world in 2011, ran at 10^{10} MIPS or 10^{16} IPS. Early personal computers in the 1980's ran at about 1 to 3 MIPS, while the UNIVAC 1 completed approximately 2000 IPS. Table 9.1 compares the run times for different machines as a function of matrix size. Figure 9.1 shows the experimentally determined calculation times on the computer that was used to write this book.

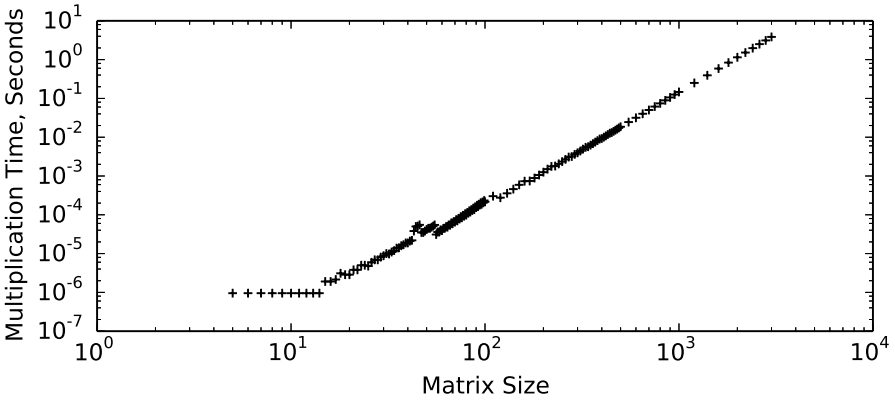
The number of basic operations or lines of code executed tell us how well an algorithm will **scale** with the size of the data set in a machine independent manner. Matrix multiplication scales proportionally to n^3 (for large n), since as n becomes large, $n^3 \gg n^2$ and therefore

$$2n^3 - n^2 \sim 2n^3 \sim O(n^3) \tag{9.2}$$

Table 9.1: Comparison of run times for matrix multiplication different machines.

n	Univac I	Early PC	Haswell	K Computer
10	950 mS	950 μS	6.3 nS	0.02 pS
100	16.6 min	995 mS	6.6 μS	20 pS
300	7.5 hrs	27 sec	180 μS	540 pS
1000	11.6 days	16.6 min	6.66 mS	20 nS
10 000	31.7 years	11.6 days	6.66 sec	20 μS
100 000	31709 years	31.7 years	1.85 hrs	20 mS
1 000 000	3.2×10^7 years	31709 years	11 weeks	20 sec

Figure 9.1: Time to multiply to two $n \times n$ matrices on an Intel Core i7 3770K machine running at approximately 100,000 MIPS using Numpy for array multiplication.



and we say that "matrix multiplication scales at big-Oh of n^3." This means that when we double the size of the input, it takes $2^3 = 8$ times as long to complete the program.

Definition 9.1. Big Oh

We say that an algorithm scales as $O(f(n))$ if there exists some constant c and some number N such that $cf(n)$ is an upper bound on the number of steps where the input size is n for all $n > N$

Definition 9.2. Little oh

We say that an algorithm scales as $o(g(n))$ if there exists some constant c and some number N such that $cg(n)$ is a lower bound on the number steps when the input size is n for all $n > N$.

In fact, we will mostly just look at big-Oh because big-Oh gives us a worst case scenario. As we know from the study of limits in calculus, its really only the highest order term

that dominates. Since matrix multiplication takes $2n^3 - n^2$ steps, then we know that for really large n, the $2n^3$ term is a lot more important than the n^2 term. Thus matrix multiplication, for example, scales with $O(n^3)$.

Now that we have a metric to use for measuring the speed of algorithm, the natural question to ask is what are good values for the metric, and what are bad values for the metric. If we start naively implementing a lot of the existence proofs from calculus or number theory, it will turn out that our calculations will take a very, very long time to run. It turns out that in very many cases we can do a lot better than these naive implementations.

- $O(n)$ - Find the largest number in a list of unsorted numbers by examining every number in the list, or looking up a word in the dictionary by starting at A and proceeding, one by, one, until we find the word we want. We say that these algorithms scale **linearly**.
- $O(\log n)$ - Look up a word in the dictionary using a **binary search**. These algorithms are a lot faster than linear algorithms. We say that these algorithms scale **logarithmically**.
- $O(n^2)$ - Any algorithm that looks at all pairs of numbers in a set of length n. **Naive sorting algorithms** scale **quadratically**.
- $O(n \log n)$ - These are faster than $O(n^2)$ but slower than $O(n)$. Examples include **efficient sorting algorithms**.
- $O(n^3)$ - An algorithm that looks at all triples of elements in a set of size n, like **matrix multiplication**, is said to scale **cubically**
- $O(a^n)$ - An algorithm that requires generating all possible subsets of any length scales **exponentially** (here a is a fixed constant, usually 2). These algorithms are excruciatingly slow.
- $O(n!)$ - Generate all possible permutations of n items. These are even worse than exponentially slow algorithms.

Really hard problems typically fall into the last two cases. To understand this combinatoric growth, consider the following table. It shows the expected time for a Haswell 5960 to complete the specified number of operations. The scaling of an algorithm can be seen by following a column in this table. For example, to see how an $O(n^3)$ algorithm scales with n, follow the values under the n^3 column.

n	$\log(n)$	n	$n \log n$	n^2	n^3	2^n	$n!$
20	14 pS	67 pS	290 pS	1.3 nS	27 nS	3.5 μS	91 D
30	16 pS	100. pS	490 pS	3. nS	90. nS	3.6 mS	2.8×10^{13} Y
50	19 pS	170 pS	940 pS	8.3 nS	417 nS	1.0 H	3.2×10^{45} Y
70	20 pS	230 pS	1.4 nS	16 nS	1.1 μS	130 Y	1.3×10^{81} Y
100	22 pS	330 pS	2.2 nS	33 nS	3.3 μS	1.3×10^{11} Y	9.9×10^{138} Y
1000	33 pS	3.3 nS	33 nS	3.3 μS	3.3 mS	1.1×10^{282} Y	4.3×10^{2548} Y
10^6	66 pS	3.3 μS	66 μS	3.3 S	39 D	10^{301011} Y	9×10^{5565689} Y

Here S=seconds; H=hours; D=days; and Y=years. As you can see, the really hard problems in the last two columns scale really poorly. Exponential and factorial order algorithms should be avoided if at all possible – even a modest $n = 30$ calculation that expands all permutations (O(n!)) will take longer than the known age of the universe to complete.

Exercises

1. Suppose algorithm f_1 has complexity $100,000n + 0.01n^2$, while algorithm f_2 has complexity $0.1n + 1000n^2$. If you are processing 50 data records, which algorithm is more efficient? 100 data items? 1000 data items?
2. Suppose an algorithm that has complexity $\sim 10n^2$ takes 30 seconds to process 1000 data records. How long will it take to process 10,000 records? One million records?
3. Prove that for a sufficiently large data set, an algorithm that has $O(e^x)$ will always take longer to run than an algorithm that has $O(x^n)$, regardless of the value of n.
4. (Requires chapter 13). Estimate the complexity of the implementation of the Babylonian algorithm used in example 13.2.
5. (Requires chapter 19). Estimate the complexity of algorithm 19.1 for the dot product.
6. (Requires chapter 19). Estimate the complexity of algorithm 19.2 for matrix multiplication.

Part II

Hacking in Python

Part II is your *Hitchhiker's Guide to the Pythonverse*.

Chapters 10 through 30 cover each of the major features of the Python language. These chapters are meant to be read more-or-less in sequence, so if you jump into a later chapter, you will likely find yourself referring back to earlier chapters.

Since the Python language is so extensive, much of it is contained in libraries. There is not enough space in a single book to fit in all of the details, so most of this material is merely summarized and tabulated. A comprehensive index to the tables can be found on pages v and vi.

You should read the text at the beginning of each chapter to become familiar with the capabilities of Python. Then, in each chapter, you should at least skim the tables to get an idea of how much has already been implemented for you. For further details, and to learn about the myriad of unmentioned and unmentionable possibilities not discussed herein, you are referred to the extensive and ever-changing online documentation.

Chapter 10

Identifiers, Expressions & Types

Identifiers

An **identifier** is the name of a variable (or other object in a computer program). In addition to variables, identifiers are the names of other things, such as functions (e.g., the sine and cosine functions in mathematics). In Python, identifiers are made up of any combination of the letters **A–Z**, **a–z**, **0–9**, and the underscore character _. A variable must begin with a non-numeric character, i.e., a letter or the underscore. **Identifiers are case sensitive**. Thus the variables **abc**, **ABC**, and **aBc** are all different.

Identifiers

- **Must** start with either a letter or an underscore character **"_"**
- **May not** start with a number
- **May** contain both letters, digits (**"0"..."9"**), and underscores
- **May** contain both upper (**"A"..."Z"**) and lower case (**"a"..."z"**) letters
- **Is** case sensitive
- **Should** not be the same as any keyword

There is a special set of identifiers in Python called **keywords** that are predefined. Any identifiers that you define should be different from all keywords. Keywords must be spelled precisely correctly, or they will not work properly.

Table 10.1. Python Keywords

and	class	elif	finally	if	lambda	print	while
as	continue	else	for	import	not	raise	with
assert	def	except	from	in	or	return	yield
break	del	exec	global	is	pass	try	

You can always access a list of the keywords inside Python:

```
In [1]: import keyword

In [2]: print(keyword.kwlist)

['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

Some additional forms of identifiers have special meanings and you should avoid these formats for your variables.

1. **The single underscore character**: _ is used (internally) by Python shell to store the result of the last calculation.
2. **Identifiers surrounded by double underscores**: `__*__`. (Here `*` represents a **wild card** character, which means (in this case) that it can be replaced by any valid identifier.) This format indicates that the identifier is defined by the system. The Python language reference says the following: "Any use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning."
3. **Identifiers preceded by a double underscore**: `__*` are private to the class in which they are defined. (Classes will be discussed in chapter 30.)

Students are often confused between **identifiers** (the names of variables) and **string literals**. Strings will be discussed in chapter 16. A string literal is a string with a specific value. You can tell the difference between the two because strings are always surrounded by quotes. Thus `'ABC'`, `"ABC"`, and `"""ABC"""` all are string literals, but `ABC` is an identifier.

Expressions

An **expression** is anything that evaluates to a **value** in Python. It may be a combination of values, operators, and variables. The **types** of the variables that are combined must be compatible, and the type of the variable on the left-hand side of the assignment is determined by the result of the evaluation. The `type` function can be used to tell us the type of a variable.

In `In[3]`, we add two integers; the result is an `int`.

```
In [3]: 5+7

Out[3]: 12

In [4]: type(12)

Out[4]: int
```

In the traditional Python shell the types are reported as, e.g., `<type 'int'>`.

In `In[5]`, we add an integer and a floating point variable (type `float`); the result is a `float`. Floats are used to represent real numbers (numbers with decimal points).

```
In [5]: 5+7.0

Out[5]: 12.0

In [6]: type(12.0)
```

```
Out[6]: float
```

If the types are incompatible, a **TypeError** will be reported and a traceback message will be printed.

```
In [7]: 5+"fred"
```

```
---------------------------------------------------------------
TypeError                       Traceback (most recent call last)
<ipython-input-7-c1666f073668> in <module>()
----> 1 5+"fred"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the traditional shell, the message referencing input 7 would be replaced with

```
File "<stdin>" line 1, in <module>
```

In a full program or a chunk of code in an iPython notebook it would refer to the line number of the code.

The expression **"fred"** has a **str** type, meaning it is string. Strings are used to represent text in computer programs (chapter 16). The message tells us that we are trying to apply the "**+**" operator between an integer and a string, but the operator is not defined on this combination of objects. We are allowed **str + str** (concatenation) and we are allowed **int + int** (addition), but not **int + str**.

Numbers in Expressions

Python supports four types of numbers: **integers**, **long integers**, **floating point numbers**, and **complex numbers**. Long integers are integers that are not limited to machine size (e.g., 64 bits). In Python 3, all integers are long. Floating point numbers represent anything with a decimal point. Complex numbers represent numbers of the form $a + bi$, where $a, b \in \mathbb{R}$ and $i^2 = -1$. (A review of complex numbers is given in appendix A.) Note that in Python, the imaginary square root of -1 is represented by the symbol **j** and not the "expected" symbol **i**.

```
In [8]: x=14.0; y=14; z=14L
```

```
In [9]: (x==y, x==z, y==z)
```

```
Out[9]: (True, True, True)
```

Integers may be expressed in decimal, octal, hexadecimal, or binary. In Python 3, the **L** is omitted. (If you attempt to type the **14L** in Python 3, like in line **In [8]** above, you will get an error message.)

Real numbers are represented as floating point numbers, called **float**'s in Python. Real numbers that are equal to integers are not integers, i.e., they are stored differently and may take up a different amount of space. The **e** and **E** formats are used for scientific notation, much like the EE key on calculators. **1234.5**, **1.2345E3**, and **1.2345e3** all represent the same number 1.2345×10^3.

Data Types in Python

The expression **data type** refers to the way a computer language represents or stores its data in a computer. A computer language typically has several different data types that are used to represent things like integers, real numbers, and strings. Sometimes, when different data types can be used in combination with one another (like real numbers and integers) these types are sub-classified in such a way to allow either easier conversion during computer programs.

To identify a variable's data type, use the function **`type`**.

```
In [10]: (type(x), type(y), type(z))

Out[10]: (float, int, long)
```

To determine if a variable has a particular data type, use the built in function **`isinstance`**.

```
In [11]: isinstance(x,int), isinstance(x,list), isinstance(x,float)

Out[11]: (False, False, True)
```

Python supports a variety of data types natively. These can be roughly classified as follows: **Numeric** types (such as integers, floats, complex numbers, and booleans (chapter 12)); **Sequential** types (particularly lists, strings, and tuples; see chapters 14, 15, and 16); **Mapping** types (dictionaries; see chapter 25); and **Sets** (chapter 17). If you need something more sophisticated, you can also define your own **class** (chapter 30).

Table 10.2. Classification of Built-in Data Types in Python

Category	Specific Types[a]
Numeric	**`int`, `float`, `complex`, `boolean`, `xrange`**[b], **`unicode`**
Sequential	**`list`, `str`, `tuple`**
Sets	**`set`**
Mapping	**`dict`**
Specialized	**`None`, `file`, `Ellipsis`**
Objects	User defined classes

[a] This list is not comprehensive. In particular, the standard library has a lot of additional data types, such as **`datetime`**, which is used to represent (surprise) dates and times.
[b] Python 2 only.

Integers

Integers are represented by the data type **`int`**. Unlike most computer languages, where the value of an integer is limited by the machine word size, there is no limit on the size of an integer in python 3.[1]

[1] In python 2.7, values were typically limited to between -2,147,483,648 and 2,147,483,647 (on 32 bit machines) and between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 on 64 bit machines,

Expressing Integers

decimal	Any sequence of decimal digits such as `7351` or `129`.
long	Any sequence of decimal digits followed by an `L` or `l` such as `7351L` or `1321`. (Use of the lower case `l` is not recommended because it can be confused with the number 1.) Not used in python 3.
binary	Begin with `0b` or `0B` (number zero, followed by letter b or B), followed by any sequence of ones (`1`) and zeros (`0`), such as `0b101101`, which represents the decimal number 45.
octal	Begin with `0O` or `0o` (number zero followed by letter o or O), followed by any sequence of octal digits (`"0".."7"`), such as `0o42763`, which represent the decimal integer 17907. (The use of the upper case O is not recommended because it can be confused with the number 0).
hex	Begin with `0x` or `0X`, followed by any sequence of hexadecimal "digits", which are `"0".."9"` and `"A".."F"` (or `"a".."f"`). Hex digits are not case sensitive. An example is `0x5ab4c`, which represents the decimal integer 371532.

In python 2, if you attempt to create an integer that is larger than allowed, e.g., by adding one to the largest possible number, your result into a **`long`** integer. There is no limit to the size of **`long`** integers. All integers in python 3 are long and the terminal **`L`** is omitted.[2]

```
In [3]: y=sys.maxint+1
```

Python

```
In [4]: y

Out[4]: 9223372036854775808L

In [5]: type(y),type(sys.maxint)

Out[5]: (long, int)
```

Notice that the **`long`** integer has the letter L appended to the end of it. Otherwise, the conversion is completely transparent. In Python 3 this distinction is hidden and the **`long`** data type is removed, so that everything just looks like an **`int`**.

Floating Point Numbers

Floating Point numbers are binary approximations to real numbers. Floating point numbers are all stored using the IEEE 754 64-bit double precision standard (page 52), regardless of what type of hardware you have (even on 32-bit machines), so you can store

and larger integers were denoted by appending an upper case L to the end of the string of digits. The L is no longer permitted in python 3.

[2]Since there is no maximum integer in python 3, there is no value **`sys.maxint`** in python 3.

numbers as small in magnitude as 10^{-308}, and as big as 10^{308}, each with approximately 17 digits of precision. If you need more precision than this you can use the arbitrary precision data types provided in Numpy.

Integer division is handled differently in python 2 and python 3.

In python 3, the result is pretty intuitive. If you divide **`3/2`**, the result is automatically converted to 1.5. In python 2, integer division was performed, and the result was zero. To perform integer division in python 3, use a double-slash. Hence **`7//6`** will return **`1`**, while **`7/6`** will return **`1.1666666666666667`**.

Expressihg Floating Point Numbers

decimal notation	Must have a decimal point, such as 3.0 or 4.7
scientific notation	Use the letter **`E`** or **`e`** to specify the exponent. For example, 3E-13, 7.2E1, and 4.746e0 represent 3×10^{-13},7.2×10^{1}, and 4.746×10^{0} as floating point numbers.

Complex Numbers

Complex numbers are stored using the **`complex`** data type, which really stores a pair of floating point numbers. For a review of the properties of complex numbers, see appendix A. The imaginary number $i = \sqrt{-1}$ is represented in Python with the letter **`j`**. Other imaginary numbers such as $27.5i$ are represented by ending a floating point number with the letter **`j`**, as in **`27.5j`**. No space is allowed between the number and the letter **`j`**. A complex number is written as the sum of a floating point number and an imaginary number. If the letter **`j`** is used by itself it is assumed to be a variable, so to represent the complex number i you would write **`1j`**. In a notebook,

```
In [1]: from cmath import sqrt
        x=1+2.5j
        y=3-7.2j
        z=x*y
        z

Out[1]: (21+0.2999999999999998j)

In [2]: sqrt(z)

Out[2]: 4.582692589942323+0.03273184859250788j)
```

The `None` Data Type

Python uses the identifier **`None`** to represent the absence of a value. Functions that do not have **`return`** statements, or have **`return`** statements without arguments, return the value of **`None`**. The argument of a function without arguments, as in **`f()`**, is **`None`**. The type of the symbol **`None`** is called **`NoneType`**, and there are no other symbols in Python with this data type.

Table 10.3. Arithmetic Operators

Operator	Description	Examples
`+`	Addition	`3 + 7` → `10` `5+6.4` → `11.4`
`-`	Subtraction	`10-6` → `4` `10.7 - 4.3` → `6.4`
`*`	Multiplication	`5*4` → `20` `5.3*4.6` → `24.38`
`/`	Division	`12/5` → `2` (Python 2) `12/5` → `2.4` (Python 3) `12.0/5` → `2.4`
`//`	Floor	`13 //5` → `2` `13.0 //5` → `2.0`
`%`	Modulo	`13 % 5` → `3` `13.0 % 5` → `3.0`

`**`	Exponentiation	`2**10` → `1024`
		`2**.5` → `1.4142135623730951`
		`2**(1/2)` → `1`

Arithmetic Evaluation in Python

Arithmetic operations are summarized in table 10.3. For the most part (i.e, when the operands are numbers) these operators work pretty much as you might expect from mathematics. The order of operations is crucial to correct evaluation of an expression. An acronym that may help you remember the default order of evaluation is **PEMDAS**:[3] **Parenthesis**, **Exponentiation**, **Multiplication/ Division**, **Addition/Subtraction**.[4] See table 10.4.

In any expression, the order of operations is important. Whenever there is any doubt, use parenthesis. If you don't, the operation may not evaluate to what you think it is going to evaluate to.

To see how operator precedence works in Python, consider the following examples. If there are no parenthesis, the expression is evaluated from left to right.

[3] Traditionally, Please Excuse My Dear Aunt Sally. In street slang, Dear Aunt Sally has become Dope Ass Swag. In other parts of the English speaking world, alternative acronyms are used: BEDMAS (B for brackets); BODMAS (O for orders or powers); BIDMAS(I for indices); and BOMDAS. The order of MD really doesn't matter, because operations with equal precedence are evaluated left to right (see table 10.4).

[4] This is short for the full acronym, **PEUMDAShAOCEAIML**. Gesundheit. Most people just use parentheses. And maybe a little nasal decongestant from time to time as needed.

Table 10.4. Operator Precedence

Lower numbers mean higher precedence. Higher precedence operators are evaluated before lower precedence operators. Operators on the same line have equal precedence and are evaluated in an expression from left to right.

	Operator		(Description)
0.	`()`	P	(parenthesis)
1.	`**`	E	(exponentiation)
2.	`~`,`+`, `-`	U	(unary plus and minus)
3.	`*`, `/`, `%`, `//`	MD	(multiplication and division)
4.	`+`, `-`	AS	(addition and subtraction)
5.	`>>`, `<<`	Sh	(shifting)
6.	`&`	A	(bitwise and)
7.	`|`, `^`	O	(bitwise or, bitwise exclusive or)
8.	`<`, `>`, `<=`, `>=`	C	(comparison)
9.	`==`, `!=`, `<>`	E	(equality, inequality)
10.	`=`, `+=`, `-=`, `%=`, `/=`, `//=`, `*=`, `**=`	A	(assignment, set equal to)
11.	`is`, `is not`	I	(identity)
12.	`in`, `not in`	M	(membership)
13.	`not`, `or`, `and`	L	(logical)

```
In [1]: 5.0/7.0/3.0/12.
```

```
Out[1]: 0.019841269841269844
```

This result is the same as if we had typed the following key sequence into a calculator:

5 / 7 = / 3 = / 12 =

The calculator treats everything like a floating point number, so this works just fine. Adding parenthesis tells to calculate 3 / 12 = first, and so on, starting from the right hand side.

```
In [2]: 5.0/(7.0/(3.0/12.0))
```

```
Out[2]: 0.17857142857142858
```

Adding parenthesis from the left tells us to calculate 5 / 7 = first, then hit / 3 =, and then hit / 12 =. We get the same answer as we did from **In[2]**.

```
In [3]: ((5.0/7.)/3.)/12.0
```

```
Out[3]: 0.019841269841269844
```

In python 2 only, the calculator analogy fails, because most calculators don't know about integers. The following is almost identical to **In[2]**, except there are no decimal points now.

```
In [4]: 5/7/3/12
```

```
Out[4]: 0
```

This gives zero in python 2 because 5/7 gives 0. In python 3, the input in input 4 will produce the same output as the floating point numbers. The equivalent python 3 code is

```
In [1]: 5/7/3/12
```

```
Out[1]: 0.019841269841269844
```

```
In [1]: 5//7//3//12
```

```
Out[1]: 0
```

Some of the operators listed in table 10.4 may look a little unusual to you, because they do not apply to normal numerical quantities. For example, **is**, **and**, **or**, and **not** refer to Boolean variables (chapter 12), and the comparison operators to Boolean expressions. The assignment operators will be discussed in chapter 11.

Function Calls in Expressions

Python expressions can include **function calls**. A function call in Python looks something like **somefunction(p1, p2, ..., pn)**, where **somefunction** is the name of the function and the **p1**, **p2**, ..., **pn** are parameters. Parameters contain information that is passed to the function. The function itself is another program that performs a computation and returns an answer. The answer is returned as the value of the function. For example, to calculate the cosine of an angle x, and assign the value to y, i.e., to find

$$y = \cos x$$

we could use the function call

```
y=math.cos(x)
```

In the function call we are invoking the function (asking python to execute the function) **math.cos** and passing it the value of the parameter **x**. Python passes the value of **x** to the program that has a name **math.cos**; then **math.cos** does its thing and sends back the value $\cos x$. We then extract this value by using the function call as part of an expression. In the case shown above, we assign the value of the number returned to the variable **y**.

There are a large number of built in functions in python that you may find yourself using frequently. Examples include the **len(u)** function, which returns the length of the list **u**; and **range(n)**, that returns a sequences integers **[0,1,2,...,n-1]**.[5] Python

[5] Unlike python 2, in python 3, the object returned by **range** is not a list, but a sequence. To actually generate a list, you must call the function list. Sequences are like calculate-on-demand lists that save space. The old **xrange** function from python 2 is the new **range** in python 3.

built in functions are summarized in table 10.5. Some of the functions in table 10.5 may look a bit mysterious right now, especially if you are new to programming. Rest assured that many of them will eventually come in handy.

In addition to the built-in functions, there are even more functions available in libraries. Some of these libraries are considered standard python libraries, which means that any python distribution should have access to them. Each library can be accessed by including a single line of code in your program that tells python that you want to use it. One example of a standard library is the **`math`** library (table 10.6), which includes the function **`math.cos`**. To do mathematics in the complex plane, there is the complex math library **`cmath`** (table 10.7). Other standard libraries that you might find useful are **`fractions`** (table 10.8), **`random`** (table 31.1), and **`operator`** (table 10.9).The standard libraries are fully documented online[6] and are far too numerous to document here. Some particular ones you might want to take a look at are **`csv`** (read and write csv files); **`os`** (operating system interfaces); **`time`**, **`datetime`**, and **`calendar`** (for working with dates and times); and **`subprocess`** (execute another program). For advanced users there are also a number of markup language libraries (HTML, XML, SGML, SAX) and GUI tools (e.g., **`tkinter`** for using **`tk`**). Before you go off coding some kind of utility function on your own (or looking for an external library) you should always check to see if python already does what you need for you.

You can also define your own functions; this is the subject of chapter 18. You will probably end up writing programs in such a way that everything (or nearly everything) is a part of a function.

[6] At https://docs.python.org/2/library/#the-python-standard-library

Table 10.5. Python Built in Functions (1 of 2)

Function	Return Value
`abs(x)`	Absolute Value, Returns $\|x\|$
`all(x)`	`True` if all the elements of `x` are `True`.
	`all([5>1, 5>99, 5<3])`→`False`
`any(x)`	`True` if any of the elements of `x` are `True`.
	`any([5>1, 5>99, 5<3])`→`True`
`bin(x)`	Converts an integer `x` to a binary string.
	`bin(99)`→`'0b1100011'`
`bool(x)`	Converts `x` to `True` or `False`.
	`bool(5+7.5/3)`→`True`; `bool(5-50/10)`→`False`
`bytearry(x)`	Converts `x` to a byte array.
	`bytearray([1,2,3])`→`bytearray(b'\x01\x02\x03')`
`callable(x)`	`True` if `x` is callable.
	`callable(math.pi)`→`False`; `callable(math.sqrt)`→`True`
`chr(i)`	Returns one-character string with ASCII code `i`.
`classmethod(f)`	A wrapper (decorator) to turn a function into a method within a class.
`cmp(x, y)`	-1, 0, or 1 depending on $x > y$, $x = y$, or $x < y$.
	`cmp(-5,4)`→`-1`; `cmp(9,0)`→`1`
`compile(s,f,m)`	Creates an object for either `exec` or `eval`.
`complex(x)`	Converts `x` into a complex number.
	`complex(5.7)`→`(5.7+0j)`; `complex(3,4)`→`(3+4j)`
`delattr(obj, n)`	Delete attribute `n` from `obj`.
`dict(x)`	Create a dictionary from `x`.
	`dict([("a",5),("b",6)])`→`{'a':5,'b':6}`
`dir()`	List of names in the current scope, or scope `x` as `dir(x)`.
`divmod(a, b)`	Quotient and remainder using long division.
	`divmod(387,123)`→`(3, 18)`
`enumerate(s)`	Converts `s` into a `next()` compatible `enumerate` object.
`eval(expr)`	Evaluates `expr` as a Python expression.
	`eval("math.e**math.pi")`→`23.140692632779263`
`exec(string)`	Parses `string` to evaluate `expr`.
	`exec("x=3; y=x+7; print 'y**2=',y**2")`→ `y**2= 100`
`execfile(fname)`	Parses the file `fname` to execute a command.
`file(name, mode)`	Constructs a file object. In most cases, it is better to use `open(name,mode)` instead of `file`. See chapter 24.
`filter(f, s)`	A list of elements from iterable `s` for which `f` is true. Equivalent to `[x for x in s if f(x)]`.
	`filter(lambda x:x%2==0, range(10))`→ `[0,2,4,6,8]`
`float(x)`	Converts `x` to a `float`.
`format(value)`	Formatted string representation.
	`format(3, "5.3f")`→`'3.000'`
`frozenset(s)`	Converts iterable `s` into a frozen set. Chapter 17.
`getattr(x, n)`	The value of attribute `n` of `x`.
`globals()`	A dictionary of the current global symbol table.
`hasattr(x, n)`	`True` if `x` has attribute `n`. See chapter 30.
`hash(x)`	Hash value of `x` if it exists.

Table 10.5. Python Built in Functions (2 of 2)

`help(x)`	Invokes interactive help system.
`hex(x)`	Converts an integer to hexadecimal string.
	`hex(99)`→`'0x63'`
`id(x)`	A unique and persistent integer associated with `x`.
`input(prompt)`	Like `eval(raw_input(prompt))`.
`int(x)`	Converts `x` to an integer.
`isinstance(x, type)`	`True` if `x` is data type `type`. See chapter 30.
	`isinstance(3.14159, float)`→`True`
	`isinstance("wind speed", int)`→`False`
`issubclass(A, B)`	`True` if `A` is a subclass of `B`. See chapter 30.
`iter(x)`	Converts `x` to an iterator.
`len(s)`	Length of `s`, a string or list.
`list(x)`	Converts iterable `x` into a list.
	`list("monty")`→`['m', 'o', 'n', 't', 'y']`
`locals()`	Local symbol table.
`long(x)`	Converts `x` into a long integer.
`map(f,s)`	See page 96.
`max(arguments)`	Maximum value of arguments or iterable.
`min(arguments)`	Minimum value of arguments or iterable.
`oct(x)`	Converts integer `x` into octal.
`open(fname, mode)`	Opens file `fname`. See chapter 24.
`ord(c)`	Unicode integer representing character `c`.
`pow(x,y)`	Power, x^y; same as `x**y`.
`print(x)`	Prints stuff. Parens. are optional in version 2.
`range(i, j, k)`	See page 121.
`raw_input(prompt)`	Writes `prompt` to terminal, waits for input, which is converted to a string.
`reload(module)`	Reloads a previously imported package.
`repr(x)`	A string with a printable representation of an `x`.
`reversed(s)`	Reverses the iterator `x`.
`round(x)`	Floating point round-off.
	`round(math.pi,4)`→`3.1416`
`reduce(f, s)`	See chapter 27, page 234,
`setattr(X, n, val)`	Sets a value of an attribute. See chapter 30.
`sorted(x)`	Sorts an iterable `x`.
	`sorted([5,8,1,7,3])`→`[1, 3, 5, 7, 8]`
`staticmethod(f)`	A static method for a function `f`.
`str(x)`	Convert `x` to a string.
`sum(s)`	The sum of the elements in the iterable `x`.
	`sum([3.5,7.2, 9.6, -4.2,0, 3.6])`→`19.7`
`tuple(s)`	Converts `s` to a tuple.
`type(x)`	The type of `x`.
`unichr(i)`	Unicode character with code given by integer i.
`unicode(x)`	Convert `x` to a unicode string.
`vars()`	Same as `locals()`.
`xrange(i,j,k)`	See page 115. Python 2 only.
`zip(x,y,..)`	See page 122.

The Python standard `math` library

A small core of standard mathematical functions are included in the **`math`** library (table 10.6). These all operate nominally on **`float`**. For equivalent operations in the complex plane use the **`cmath`** library (table 10.7). Like any library, it can be included in whole or in part. To include the entire library put the statement, use the wild-card (asterisk) notation.[7] Then you can use the functions directly. For example, to calculate $\sqrt{x}$, type **`sqrt(x)`** into your code.

```
In [1]: from math import *
        cos(pi/2), sin(pi/2), pi**e

Out[1]: (6.123233995736766e-17, 1.0, 22.45915771836104)
```

Sometimes you will want to have multiple libraries loaded that have functions with the same names. For example, both **`math`** and **`numpy`** have **`sqrt`** functions (as well as the trig functions) in common. To distinguish between these libraries, we would instead using a simple package import. The entire library is still available to your program.

```
In [1]: import math
        math.cos(math.pi/2), math.sin(math.pi/2), math.pi**math.e

Out[1]: (6.123233995736766e-17, 1.0, 22.45915771836104)
```

Then to calculate $\sqrt{x}$ using the **`math`** library you would have to put use the expression **`math.sqrt(x)`** instead of just **`sqrt(x)`**. If **`cmath`** and **`numpy`** are also loaded in this way, then you could use their square root functions as **`cmath.sqrt(x)`** and **`numpy.sqrt(x)`**.

```
In [2]: import numpy
        print (numpy.sqrt(4), math.sqrt(4), numpy.sqrt([1,4,9]))

        (2.0, 2.0, array([ 1.,  2.,  3.]))
```

This way you are able to tell Python which **`sqrt`** function to use. An error would have been generated if you had tried to calculate the square root of the list **`[1,4,9]`** with **`math.sqrt`**, because **`math.sqrt`** is not defined for lists. It is only defined for numbers.

[7] This practice is discouraged, however, because it pollutes the name space. For example, following **`from math import *`** with **`from numpy import *`** could lead to unexpected results as many of the functions have the same name.

Table 10.6. Python `math` Library (1 of 2)

Function[a]	Return Value
`acos(x)`	arccos(x) in radians.
	`round(pi/acos(sqrt(3)/2),10)`→`6.0`
`acosh(x)`	Hyperbolic arc cosine, arccosh(x).
`asin(x)`	arcsin(x) in radians.
`asinh(x)`	Hyperbolic arc ine, arcsinh(x).
`atan(x)`	arctan(x) in radians.
`atanh(x)`	Hyperbolic arc tangent, arctanh(x)
`atan2(y,x)`	arctan(y, x) in radians, in correct quadrant, between $-\pi$ and π.
	`round(degrees(atan2(-1/2, sqrt(3))),10)`→`-30.0`
`ceil(x)`	Integer ceiling as a float: $\lceil x \rceil$.
	`ceil(4.2)`→`5.0`; `ceil(-1.9)`→`-1.0`
`copysign(x,y)`	x with the sign of y.
	`copysign(3,-42.7)`→`-.0`
`cos(x)`	$\cos x$, where x is in radians.
`cosh(x)`	Hyperbolic cosine, $\cosh x$
`degrees(x)`	Converts x from radians to degrees.
	`degrees(pi/2)`→`90.0`
`e`	e ($\approx 2.71828\cdots$)
`erf(x)`	Error function $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
`erfc(x)`	Complementary error function `erfc(x)`→`1-erf(x)`
`exp(x)`	e^x (exponential).
`expm1(x)`	$e^x - 1$ (exponential minus 1).
`fabs(x)`	Absolute value of a floating point number $\lvert x \rvert$.
`factorial(n)`	Factorial $n!$; n must be a non-negative integer.
	`factorial(6)`→`72` `factorial(23)`→`25852016738884976640000L`
`floor(x)`	Integer floor as a float: $\lfloor x \rfloor$.
	`floor(37.2)`→`37.0`
`fmod(x,y)`	Floating point modulo division using the platform C library.
`frexp(x)`	Mantissa an exponent (m, e) such that $x = 2^e m$.
	`frexp(1985)`→`(0.96923828125, 11)` `frexp(256)`→`(0.5, 9)`
`fsum(s)`	Floating point sum of iterable `s`.
`gamma(x)`	Gamma $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$; $\Gamma(n) = (n-1)!$, $n \in \mathbb{Z}^+$.
`gcd(x,y)`	Greatest common divisor. New in python 3.
`hypot(x,y)`	Length of hypotenuse of a right triangle. $\sqrt{x^2 + y^2}$
	`hypot(-3.4, 4.2)`→`5.403702434442518`
`isinf(x)`	Checks if `x` is $\pm\infty$.
`isnan(x)`	Checks if `x` is not a number.
`ldexp(x,i)`	Returns $x2^i$; the inverse of `frexp`.
	`ldexp(.5,9)`→`256.0`
`lgamma(x)`	$\ln \lvert \Gamma(x) \rvert$

[a] For more details see https://docs.python.org/3/library/math.html.

Table 10.6. Python `math` Library (2 of 2)

Function	Return Value
`log(x,b)`	$\log_b(x)$; if `b` is omitted, returns $\ln x$.
	`log(81,3)`→`4.0`
`log1p(x)`	$\ln(1+x)$
`log10(x)`	$\log_{10}(x)$
`modf(x)`	Returns (f, i) where f is the fractional and i is the integer part of `x`, as floating point numbers.
	`modf(8.246)`→`(0.246, 8.0)`
`pi`	π.
`pow(x,y)`	x^y. Converts its arguments to float first.
`radians(x)`	Converts x from degrees to radians.
`sin(x)`	$\sin x$, where x is in radians.
`sinh(x)`	$\sinh x$.
`sqrt(x)`	$\sqrt{x}$
`tan(x)`	$\tan x$, where x is in radians.
`tanh(x)`	$\tanh x$.
`trunc(x)`	Truncates a real number to an integer.

Table 10.7. Python `cmath` (complex math) Library

Function[a]	Return Value
`acos(z)`	$\arccos z$. Branch cuts along $(-\infty, -1)$ and $(1, \infty)$.
`acosh(z)`	$\operatorname{arccosh} z$. Branch cut along $(-\infty, 1)$.
`asin(z)`	$\arcsin z$. Branch cuts along $(-\infty, -1)$ and $(1, \infty)$.
`ainsh(z)`	$\operatorname{arcsinh} z$. Branch cuts $(-j\infty, -j)$ and $(j, j\infty)$.
`atan(z)`	$\arctan z$. Branch cuts $(-j\infty, -j)$ and $(j, j\infty)$.
`atanh(z)`	$\operatorname{arctanh} z$. Branch cuts $(-\infty, -1)$ and $(1, \infty)$.
`cos(z)`	$\cos z$
`cosh(z)`	$\cosh z$
`e`	e
`exp(z)`	e^z
`isinf(z)`	Tests if either the real or imaginary part of z is infinite.
`isnan(z)`	Tests if either the real or imaginary part of z is not a number.
`log(z,b)`	$\log_b(z)$. b is optional. Branch cut $(-\infty, 0)$.
`log10(z)`	$\log_{10}(z)$. Branch cut $(-\infty, 0)$.
`phase(z)`	Phase of a complex number.
`pi`	π.
`polar(z)`	(r, ϕ); $r = \lvert z \rvert$ and ϕ is the phase of z.
`rect(r,phi)`	Complex number with given magnitude and phase.
`sin(z)`	$\sin z$
`sinh(z)`	$\sinh z$
`sqrt(z)`	$\sqrt{z}$. Branch cut on the negative real axis.
`tan(z)`	$\tan z$
`tanh(z)`	$\tanh z$

[a] For more details see https://docs.python.org/3/library/cmath.html.

The Python standard `fractions` library

The fraction library (table 10.8) allows you to deal directly with **rational numbers** as ratios of integers. A rational number r, represented *mathematically* by the fraction $r = m/n$, where m and n are integers, can be represented by the Python expressions

```
fraction(m,n)
```

or

```
fraction(m/n)
```

Normal arithmetic operations may be performed on fractions; the computation will be performed exactly and the answer is returned as a fraction. For example

```
In [1]: from fractions import Fraction
        x=Fraction(3,4)
        x+y

Out[1]: Fraction(13, 8)

In [2]:  x*y

Out[2]: Fraction(21, 32)
```

In mixed mode operations with integers, the integer is converted to a fraction:

```
In [3]: 1+Fraction("5/7")/Fraction(3,8)

Out[3]: Fraction(61, 21)
```

In mixed mode with floats, the fraction is converted to a float:

```
In [4]: pi*Fraction(1,2)

Out[4]: 1.5707963267948966
```

Table 10.8. Python `fractions` Library

Function[a]	Return Value
`Fraction(a,b)`	Represents a rational number a/b, where $a, b \in \mathbb{Z}$.
`Fraction("a/b")`	Same as `Fraction(a,b)`
`Fraction(x)`	Converts floating point number to Fraction.
`gcd(a,b)`	Greatest common division of $a, b \in \mathbb{Z}$. Deprecated in favor of `math.gcd(a,b)`.

[a]For more details see https://docs.python.org/3/library/fractions.html.

The `operator` library

All python operators are also implemented as functions in the **`operator`** library. For example, **`operator.add(x,y)`** means the same thing as **`x+y`**.

Table 10.9. Python standard `operator` library (1 of 2)

Function[a]	Return Value
`add(a, b)`	`a+b`
`abs(x)`	`abs(x)`
`and_(a,b)`	`a and b`
`attrgetter(attr)`	Returns a function that can retrieve attributes of an object.
`concat(s,t)`	`s+t` (concatenation)
`contains(s,x)`	`x in s`
`countOf(a,b)`	Number of times sequence `b` occurs in sequence `a`.
`delitem(s, i)`	Deletes item `s[i]` from `s`
`div(a,b)`	`a/b` when `__future__.division` is not in effect
`eq(a, b)`	`a==b`
`floordiv(a,b)`	`a//b`
`ge(a, b)`	`a>=b`
`getitem(x,i)`	`x[i]`
`gt(a, b)`	`a>b`
`iadd(a, b)`	`a=iadd(a,b)` →`a+=b` (calculation in place)
`iand(a, b)`	`a=iand(a,b)` →`a&=b` (calculation in place)
`iconcat(p,q)`	`p=iconcat(p,q)` →`p+=q` (calculation in place)
`idiv(a, b)`	`a=idiv(a,b)` →`a/=b` (calculation in place)
`ifloordiv(a, b)`	`a=ifloordiv(a,b)` →`a//=b` (calculation in place)
`ilshift(a, b)`	`a=ilshift(a,b)` →`a<<=b` (calculation in place)
`imod(a, b)`	`a=imod(a,b)` →`a%=b` (calculation in place)
`imul(a, b)`	`a=imul(a,b)` →`a*=b` (calculation in place)
`indexOf(a,b)`	Index of first occurrence of `b` in `a`.
`inv(x)`	`-x`
`invert(x)`	Same as `inv(x)`
`ior(a, b)`	`a=ior(a,b)` →`a|=b` (calculation in place)
`ipow(a, b)`	`a=ipow(a,b)` →`a**b` (calculation in place)
`irshift(a, b)`	`a=irshift(a,b)` →`a>>=b` (calculation in place)
`is_(a, b)`	`a is b`
`is_not(a, b)`	`a is not b`

[a] For more details see https://docs.python.org/3/library/operator.html.