Scientific Computation:
Python Hacking for Math Junkies
Second Edition, revised

This book is for math junkies - anyone who likes math or uses it in their daily life. It is also for anyone who wants to learn about scientific or mathematical computation quickly and easily using Python. The emphasis is on algorithms and hacking your way to solutions as quickly as possible. It is approachable by students with no prior programming experience, and yet useful for experienced scientists.

Topics include Python expressions, statements, types, lists, arrays, functions, classes, plotting, list comprehension, recursion, iPython notebooks, linear systems, computational geometry, roots, interpolation, least squares, discrete systems, differential equations, the singular value decomposition, principal component analysis, logistic regression, clustering, fractals, and chaos. Includes coverage of numpy, pyplot, and mapping with baseplot.
Over two hundred exercises.

Python Hacking for Math Junkies introduces "the art of the possible," with playful elegance.

```
from matplotlib.pyplot import *
import numpy as np
def mand(C, zmax, n):
    z = C
    for j in xrange(n):
        z=z*z+C
        if abs(z)>zmax:
            return(j)
    return(0)
def mandelbrot(xc, yc, r, pix, zmax, nmax):
    xmin = xc-r; xmax = xc + r
    ymin = yc-r; ymax = yc + r
    xvals=np.linspace(xmin,xmax,pix)
    yvals=np.linspace(ymin,ymax,pix)
    dy = (float(ymax)-float(ymin))/float(pix)
    dx = (float(xmax)-float(xmin))/float(pix)
    image=np.ones((pix,pix))
    for ix in xrange(pix):
        for iy in xrange(pix):
            z=(xmin+dx*ix)+(1j)*(ymin+dy*iy)
            image[iy,ix]=mand(z, zmax, nmax)
    return image
i=mandelbrot(.2323,-0.5345,.0025,500,3,500)
imshow(i)
show()
```

Scientific Computation:
Python Hacking for Math Junkies

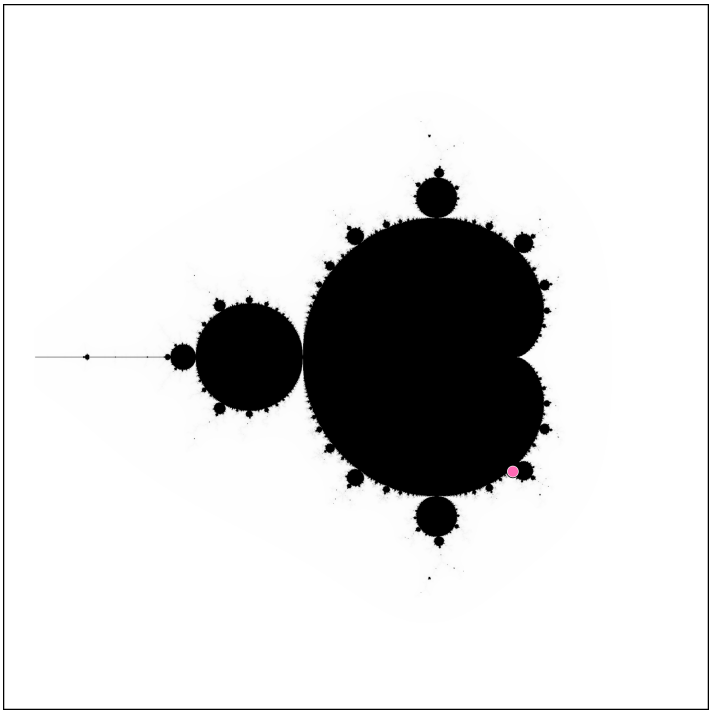
Bruce E Shapiro

Scientific Computation

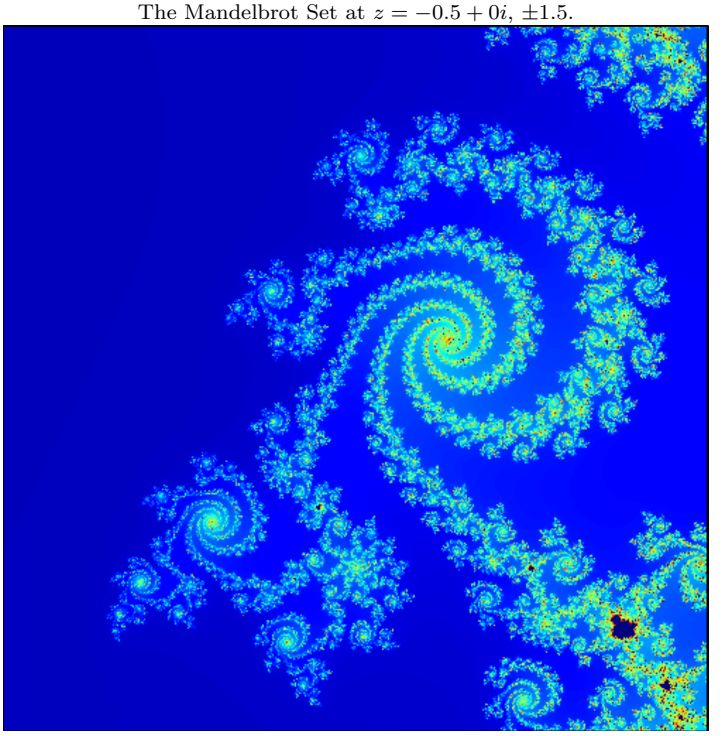
Python Hacking for Math Junkies

Second Edition, Revised (v. 2.5)

Bruce E. Shapiro



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.

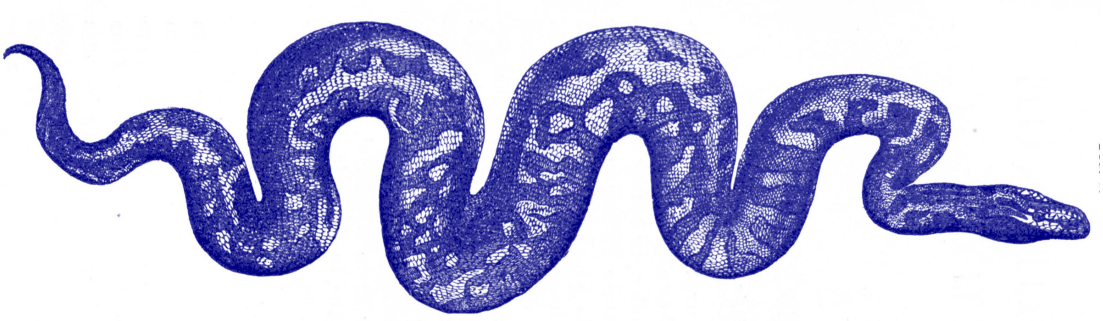
The Mandelbrot set displays the properties of fractals: it is bounded; its edge has infinite self similarity; and its boundary has an infinite length.

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 $z = 0.2323 - 0.5345i$ (Python code on back cover; 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 Lebesque Integration.

Scientific Computation



Python Hacking for Math Junkies

Second Edition, Revised

Bruce E. Shapiro

2015

Sherwood Forest

http://CalculusCastle.com

Scientific Computation: Python Hacking for Math Junkies
Second Edition, Revised
Bruce E. Shapiro, Ph.D.
California State University, Northridge
Sherwood Forest Books, Los Angeles, CA, USA

ISBN-13: 978-0692452004 / ISBN 10: 0692452001 (BW/Paper-Blue Cover)
ISBN-13: 978-0692366936 / ISBN-10: 0692366938 (BW/Paper-Orange Cover)
ISBN-13: 978-0692498552 / ISBN-10: 0692498559 (Color/Paper)
ISBN-13: 978-0996686006 / ISBN 10: 0996686002 (Hardbound)
ISBN-13: 978-0996686013 / ISBN 10: 0996686010 (Electronic)

Last Revised on: 1/17/16 (Version 2.5)

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 *Cat in the Hat* picture on the back cover was drawn by J. Fors.

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.

Please report any errors, omissions, or suggestions for improvements to bruce.e.shapiro@csun.edu. Please include the version number and revision date listed at the top of this page on all communications.

And grateful thanks to the many useful suggestions and corrections provided by readers.

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.

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

Almost everyone uses computers today. They improve our lives and produce wonderful works of art. Information is accessible and global communication is immediate. It is hard to imagine (or for some of us, remember) the days when you couldn't just look something up or chat with a friend or share your innermost thoughts with total strangers on the other side of the planet. We even have verbs for such things now: *to google*, *to skype*, *to tweet*.

The typical student uses at least three: a portable notebook computer that weighs no more than a textbook, a tablet, and a cell phone. None of these even existed before students in the class of 2016 were born. A lot has changed: ask your grandparents about AM radio, phone booths, or broadcast TV. Now when you feel an earthquake or see a fire plume or hear a siren the first place to go for information is Twitter.

Hacking and Programming

Naturally some computer literacy is expected of everyone; things like carbon copies of credit cards are mostly archaeological relics. Instant verification is here and now. Even paper and pencil are disparaged by many: "Shift or get off the pot. Seriously, it's not fair to the kids. It's tough at the outset to understand and learn all these tools, but you're doing a disservice to our students and these kid's futures if you don't."[1] Yet for many – including the teachers – computers are a source of trepidation and fear. This becomes especially true when computers are misunderstood or used improperly. And as math users, we know that the regardless of what some tech executives may want us to believe, we are still a very long way from letting go of paper and pencil.

The typical user needs to know how to run apps and keep the computer safe from infiltration. He or she need not be an expert on computer speak to do this. And as mathies, we need to know especially how we can use our computers to help us solve mathematics problems. This will involve a high-level understanding of how a computer works and what tools we can use to make it work optimally to solve our kind of problems.

It's like driving a car. You need to know to slow down for bumps and potholes and to take your car to a mechanic for regular service or when it is not working properly. You don't need to know anything about distributors or gaskets or struts (if those things even exist on cars today). Nor do you need to be an automotive engineer; you just need to know where to stick the key and what buttons to push and turn. If you are driving a truck or motorcycle, there are few differences to learn, but not so many that you need to become a mechanic (or anywhere close).

[1] L. De Cicco Remu, Director of Partners in Learning at Microsoft Canada, reported in straight.com, Stephen Hui, 15 May 2015, 4:28 PM, http://www.straight.com/life/452561/teachers-using-pens-and-paper-classroom-not-fair-students-microsoft-official-says.

Computers are similar. You need to keep your software up to date and avoid opening potentially dangerous attachments (the potholes). If it breaks, you take it in for service (the mechanic or IT guy). The people who design new software to sell you with your computer (programmers) are like the automotive engineers who design the cars. You need to know how to run their programs, not how they are designed. In fact, in most cases, the software design is kept secret (for arguable reasons).

What if you want to build your own car from scratch? Sure, you can become an engineer (computer programmer) and learn all the ins and outs of the ideal design. But what about the guy who is rebuilding an old Ferrari or motorcycle or speed boat in his garage around the corner? These individuals are true artists! They are *hackers*. They just do it.[2] They have managed to learn what they need to learn to get the job done and then have gone ahead and built their product. No bells and whistles. Just an elegant one-of-a-kind that gets the job done.[3]

Mathematical Hacking

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.

There are some commercial programs that will let you begin to do advanced mathematical computation without learning any "programming" at all. But they all have a complicated programming-like syntax or language and you will eventually need to learn this syntax if you want to anything beyond, say, solve a simple integral or perform an arithmetic calculation. So the question then becomes which programming syntax is the "best" one to learn. These syntaxes started to evolve in the middle of the twentieth century, with Fortran-like languages often being the preference of the scientific community; LISP-like languages being chosen by the artificial intelligence community; and COBOL-like languages being chosen by the data processing community. Since then, the distinctions have begun to blur and the number of languages had grown into the thousands.

Most of the early work on computation and computing grew from the *mathematical literati*. But market forces drive the industry: anything that makes money, or is exciting and glamorous, with growth potential, is what sells. These are things like business, social, and data manipulation applications. For most people, math is not glamorous. The typical "man on the street" tends to bunch math and computer types together in one big sticky mess in their brain, to be both feared and respected, but only to be approached carefully and with trepidation in matters of extreme urgency: "Please, my child needs help with advanced placement calculus" or "Why is my email so slow?" as if these subjects have any relation to one another.

Consequently, much of the beauty of computation – and what it can do for us math junkies – is largely ignored by both introductory computer science classes and introductory programming books as *irrelevant*. In industry, the "calcs" and the "squints" are expected to remain quietly in their dusty old offices and smelly laboratories (preferably on campus, so they don't have to be on the permanent industrial payroll) grinding out research reports until their specific expertise is actually requested on a need-to-know basis. Since neither undergraduate computer science nor mathematics curricula pay much serious attention to *mathematical* computing, many a promising student ends up as professional cappuccino *baristi*, substitute teachers, or insurance peddler.

[2] With apologies to Nike.

[3] 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

This book is designed rescue these math junkies and incipient mathematical literati from the dangerous shoals of computational irrelevance and confusion and lead them to the calm, clear waters of computational beauty.

Scientific Computing is not Programming

So you want to do scientific computing but have never taken a programming course, except maybe that Java 101 class you had as a freshman that was all about objects and interfaces and stuff and all you could say was wtf? huh? Or you want to do some data mining but the local college catalog says you need to take about eight courses first? Or looking at the books on Amazon it seems like you need to learn all about algorithms, databases, SQL, machine learning, cloud computing, statistics, visualization, interface design, graph theory, and so forth before you can do cull any meaningful results from your data set? And that you have learn a whole host of computer languages and styles, object oriented design, systems analysis, pair programming, and so forth.

None of that is true.

You have something that most programmers don't: love and knowledge of math.

Of course, learning as much as you can about any of those subjects (and more) will help you, but you don't need to learn about them before you start doing interesting, useful, creative, original and potentially publishable computational mathematics.

There is very little difference between writing a program to solve a problem and proving a theorem. In each case you need to produce a logical sequence of steps that takes you from input (assumptions) to output (conclusions). For most mathematicians, an understanding of the semantics, or meaning, of the statements in a computer program comes easily, once the logical analogues in mathematical analysis are understood. Usually the problem is understanding the *syntax*, and decoding of error messages.

Most 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.[4] Documentation becomes replaced by in-line comments and web pages. Copies of the source code go into open access repositories for the world to see, redact, and improve.

Students need to explore the limits of what is possible without the restrictions of a particular programming paradigm. Its the quality of their work that is important. Scientific computing is nothing like the stuff taught in introductory computer science classes.

Why Python?

I use Python as a teaching language for a number of reasons, chief among them that it is free on all operating systems. Thus students can install it on their laptops or home computers and are not chained to computer labs or expensive license agreements. Python has generic programming structures that are common to procedural, functional, and object oriented languages. If you already know another programming language, then Python is easy to learn. If you don't know how to program, then Python is free, it is completely documented online, and there are lots of books you can download for free to learn more about it.

[4] 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.

Why *this* book?

There are a lot of books on Python, but unfortunately none of them are oriented towards the mathematical literati who can't program. There are books on advanced mathematical and scientific programming for advanced programmers, and there are (far too many) non-mathematical books on programming in Python for beginners. There are books on *programming* in Python for students at just about every level of prior experience. Unfortunately, very few say anything about Numpy, plotting, or vectors, and very nearly all of them completely give up the ghost when it comes to list comprehension and lambda functions. And if you want to try to learn about Voronoi diagrams, and draw them at them same time, well, you need to buy a whole shelf full of books.

So I've – ahem – hacked out a sort of *Math Hacker's Guide to the Pythonverse.*

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.

Much of the material on Python is relegated to reference tables, rather than detailed how-to-do-its. Users are referred to the official documentation for further details. The goal of the tables is to collect some of the more relevant material so that students can focus their search into appropriate online documentation.

This book stems from my own professional experience. While some of the topics are stretched pretty far from the traditional undergraduate mathematics curriculum, I've included them for good reason. With the exception of the chapters on fractals and calculating the value of π,[5] **I have had to use virtually every single algorithm, method, hack, and technique presented in this book at some time in my professional life**.

A Note To My Students

Read Part I immediately, especially if you haven't programmed before or have been ruined by a single course in Java-based object-oriented design.

Read Part II as soon as you can, but no later than each section it is covered in class. Don't skip anything.

Read Part III as needed for reference.

Hacking is a hands-on experience. You can't expect to learn only by watching demonstrations or doing it in the lab. You have to take it home with you and practice it in your sleep. Live, think and dream in Python.

The Student's Imperative

Go forth and hack: but do no evil.

For Further Reading

Google it. Everything that's ever been done in Python is on the internet.

[5] I included both of these subjects because, well, they just sound like a lot of fun.

The Art of the Possible

"Politics is the art of the possible."[6] 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.[7]

Either way, it makes you think.

Bella does Python.

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

Notation

`Typewriter Bold` font is used to indicate single words code.

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

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

Code typed into the Python shell is enclosed in a gray box and preceded by the Python **`>>>`** prompt, which you should not type.

```
>>> print "Hello,␣World!"
Hello, World!
```

The underbracket character **`"␣"`** represents a blank space inside of a string literal.

The forward arrow symbol → is used as shorthand to represent the output of entering something in Python, as in **`print x+y`** → **`23`** is a short form of writing "if you type **`print x+y`** into the command shell now, then Python will return a value of **`23`**."

Algorithms have the following kind of format, 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

An assigment statement in an algorithm like the $x \leftarrow a$ on line 1 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.

Part I

Getting Started

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 mis-speak, 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**. Python is one of many different computer languages. There are so many different computer languages (thousands) that the Wikipedia has a list of Wikipedia pages containing lists of computer languages.[3] 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 the collection of all computer programs on a computer as providing several **layers of software** that lay on top of 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.

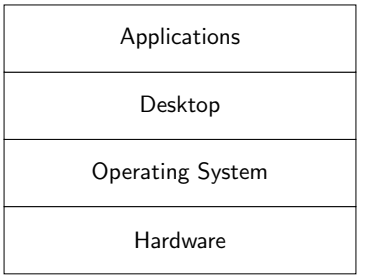
Computer programs can be broadly grouped into **application** programs and **system**

[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] As of 5 Dec 2014 there were fifteen different lists of programming languages on the "list of programming languages" page.

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.



software. **Application programs** are designed for the "end user."[4] Examples include spreadsheets, word processors, presentation programs, and web browsers. **System software** 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.

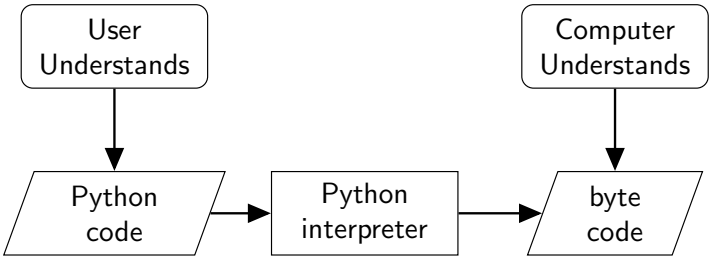
An exception is the **desktop environment**. Desktop environments control the look and feel of user interaction with the computer, through things like menus, icons, folders, task bars and launchers. Linux users have a variety of desktop environments to choose from, with names like gnome, KDE, x-windows, and so forth; many even choose to install multiple desktops on their computer and switch back and forth between them. 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.

A special class of application programs – **development software** – are tools that allow or assist users to create new computer programs. Examples are compilers and interpreters; text editors; and software development environments. The first part of this book will focus on how to use one of these tools: python.

Python is a computer language, which means it has both a **syntax** and **semantics**, as does English. But in this case, the reader of the language is not a human. 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

[4]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.

Figure 1.2: When you invoke the python interpreter to run your program, 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 and new byte code version 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.



language or byte code.

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:

```
1 >>> print 6^3
2 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.[5]. (Yes, exponentiation in Python uses the same syntax as FORTRAN!)

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."[6] 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 $\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}$."

[5] See table 11.3: 6 xor 3 = $(0110)_2$ xor $(0011)_2$ = $(0101)_2$ = 5.
[6] Ed Post (1982) *Real Programmers Don't Use Pascal.*

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.

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]Often called **data structures** or **classes**.
[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.[7]

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.

[7]This has always been popular at NASA, and you know how many disasters that's lead to.

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.

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 28 and 29, for example) before we even thought about calculating a singular value decomposition (we'll get to that, too, in chapter 40). 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 29), 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 (page 26), a single cell in an iPython notebook, or even an entire iPython notebook (chapter 5). 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 **z**. 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 9. Assignments and other simple statements are discussed in more detail in chapter 10.

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 6 and 9 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 58). 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 11.

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 15). 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 14). A **`set`** is an unordered collection of unique items (chapter 16), 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 24). Finally, you can always define your own data type with classes (chapter 29).

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 **has 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 11, 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 87), 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 11.

Functions, Classes, and Packages

Functions (chapter 18) and classes (chapter 29) 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 9).

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 12) 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 17) 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 20) 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 28).

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 10 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 programs **pip** or **easy_install**, 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**.

A list of some commonly used libraries is given in table 2.1.[3].

[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] A complete list of the Python Standard Libraries is given at https://docs.python.org/2/library/. A list of non-standard but useful scientific libraries is maintained on the wiki at https://wiki.python.org/moin/NumericAndScientific/Libraries

Table 2.1. Some Python Libraries

Some Commonly Used Standard Libraries		
math	Math	Table 9.6.
cmath	Complex Math	Table 9.7.
fractions	Rational Numbers	Table 9.8.
random	Random numbers	Table 9.9.
operator	Function equivalents of Operators	Table 9.10.
itertools	Permutations, Combinations	Table 28.1.

Other Commonly Used Libraries Mentioned in the Text		
matplotlib	Plotting	Chapter 22.
matplotlib.pyplot	Plotting	Chapter 22.
numpy	Numerical Analysis	Chapter 21.
numpy.linalg	Linear Algebra	Table 30.1.
numpy.random	Random Numbers	Chapter 34.
scipy.integrate	Numerical Integration	Tables 36.1 -36.3.
scipy.interpolate	Interpolation	Table 32.1.
scipy.spatial	Spatial Geometry	Chapter 31.
statsmodel	Statistical Analysis	Chapter 34.
Image	Python Imaging Library	Chapter 43.

Other Commonly Used Libraries (Not Mentioned in the Text)	
pandas	Data Analysis
sympy	Symbolic Mathematics
scikit-learn	Machine Learning

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: $x_{new} \leftarrow 0.5(x + a/x)$
4: $\Delta \leftarrow |x_{new} - x|$
5: $x \leftarrow x_{new}$
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:

0) **input**: gives the list of numbers that must be provided before you can begin the algorithm.

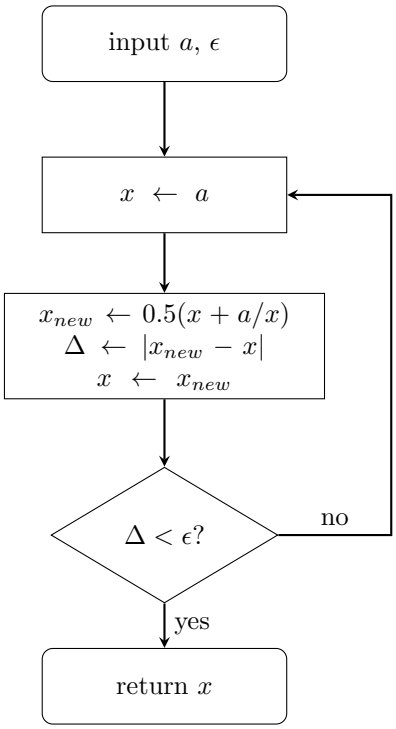
1,3,4,5) $x \leftarrow a$ means replace the symbol x with the value of symbol a. The arrow is used to indicate which direction the copying goes: the symbol at the arrowhead is changed, while the symbol or expression at the foot of the arrow is not affected. In lines 3, 4, and 5, a computation is performed first, and the result of the computation is then copied into the symbol on the left.

2...6) **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.

7) **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.

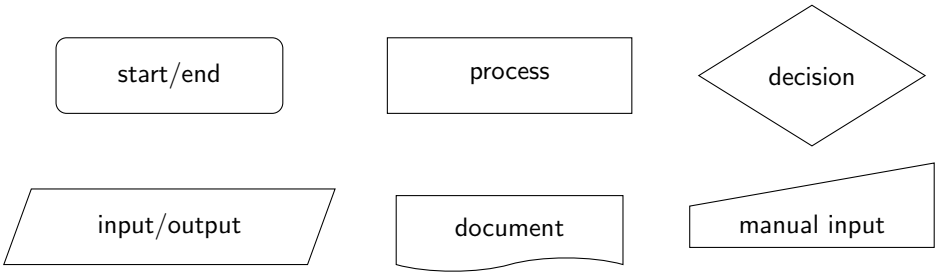
Figure 3.1: Flowchart for the Babylonian algorithm.



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

[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.



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 $p_0, p_1, p_2, \cdots$ 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

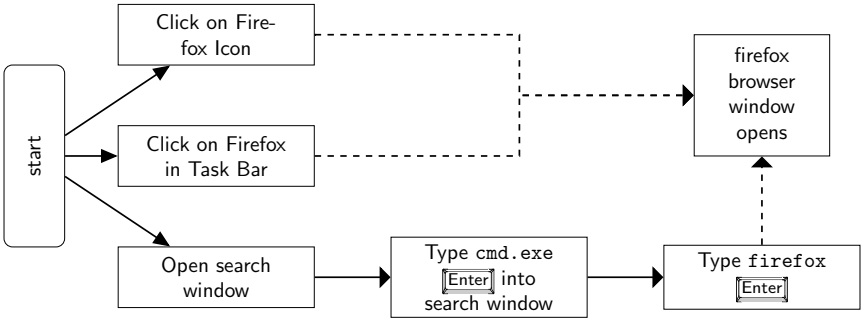
You are probably most familiar with opening your apps by **point and click**. You find the right icon and click (or double click, depending on the operating system) to open it. Variations may include some level of navigation through the file hierarchy or menus (e.g., an application or start menu); a 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 people, this will be 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 the **command shell** program on your operating system. The command shell is itself an application that you will normally open by one of the methods described in the previous paragraph. But it is a special application that allows you to run any other application on your computer *by typing in the name of the application.*[1] For example, I can open the Firefox web browser on my computer in any of the following ways (see fig. 4.1):

1. Click on the **`Firefox`** icon on my desktop.
2. Click on the **`Firefox`** icon on my launcher, dock, taskbar, or start menu (which one will depend on the operating system I am using).

[1] Sometimes a slight variation on the name of the application is required.

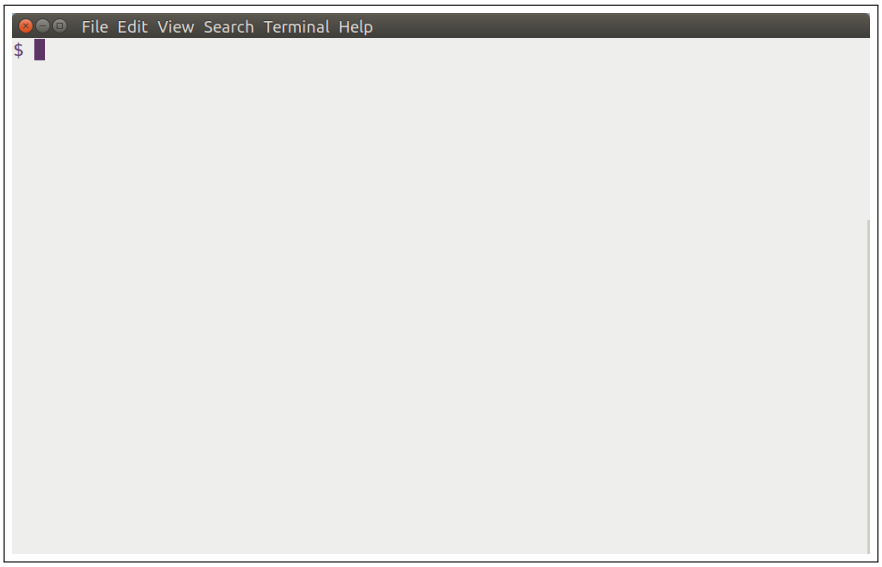
Figure 4.1: Three different ways to open the firefox web browser on Windows.



3. Open my command shell application and then type **`firefox`**, followed by the Enter key.

The name of the command shell program on will depend on your operating system. On mac and linux operating systems, it is called **`terminal`**. On windows, you can use either **`cmd.exe`** (based on the old DOS operating system) or **`powershell`** (a more advanced shell). When the command shell application is open, it will look like some variation of figure 4.2. The command shell is just an empty palette for typing in commands. All of the commands are in text, and must be entered from the keyboard.[2]

Figure 4.2: The command shell in linux. There may be some variations in colors. White writing on a black background is typically default, and you may see a copyright message at the top. In Windows the **`$`**, for example, is replaced by a **`>`**, and may be preceded by the name of the current directory.

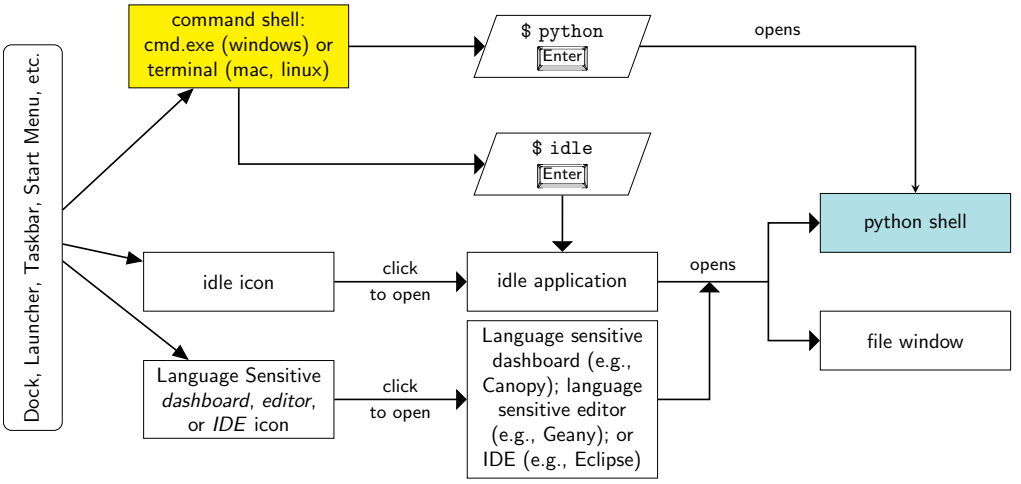


From within the command shell you can open any program on your computer by typing in the name of the program.

The program we will be using first in this book is called the **python shell**. You get to the python shell by typing **`python`** Enter in the command shell (fig. 4.3). When you do this the **prompt** character (the **`$`** or **`>`** in the command shell) is replaced by a new prompt, **`>>>`** (figure 4.4). When you are in the python shell, you can only enter python commands. You may not open other applications from this window, and you may not enter operating system commands. You may open additional command shell windows

[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 4.3: The python shell can be entered in different ways: through the command shell, through idle, through a language sensitive editor, dashboard, or interactive development environment.



if you want to do these things while the python shell is open.

Sometimes you will use an interactive development environment. The simplest one is called **`idle`**, and it comes with all python installations. When you open **`idle`**, either from the command line or from an icon, you will also see a python shell, though it will look a little bit different from the python shell you saw in the previous paragraph. Again, you can only run python commands and not execute other programs from python shell in idle. Additionally, idle has a text editor that allows you edit, save, and run programs, as described later in this chapter. Proprietary environments like anaconda or canopy have their own dashboards and programming environments, which we will not discuss further in this text.

So Let's Do it in Python

How might we implement the Babylonian algorithm Python? Lets start by opening the Python shell. First you need to open a command line program on your operating system (e.g., **`cmd.exe`** (on Windows) or **`terminal`** (on Linux or Mac)) and then type in **`python`** (see figure 4.4). Your computer might have other versions of Python installed as well, like iPython or qtPython that you can open from icons on your desktop. **Do not use these now**. You can use these later; for now you should learn the basic Python window.

In windows: open a search menu, and search for **`cmd.exe`**. This will open the command line.

On a mac: from the applications > utilities menu, look for **`terminal`**. This is the command line program.

In the **Python shell** you will see a `>>>` at the beginning of each line. This `>>>` is called a **prompt**. When you see the prompt, it means that Python is waiting for you to do something. If you see a different prompt, like `$`, then you are not in the Python shell.

To leave the Python shell and return to the **command line program** you can always type in `exit(0)`, in any operating system. In Linux you can also use Ctrl D.

We will often need both a command line window and a Python window open at the same time. To do this, open Python as described above in the first window. Then, if you are in Windows, go to the search window and select `cmd.exe` to open a second command shell. From Linux or Mac you can open a second command window from the file menu (see 4.5).

Figure 4.4: Opening the Python shell from the Windows 8 command prompt.

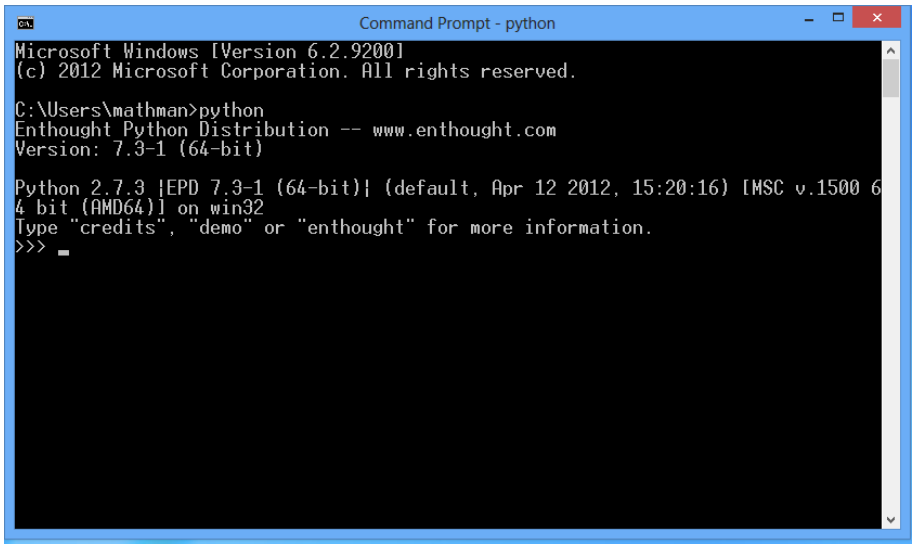
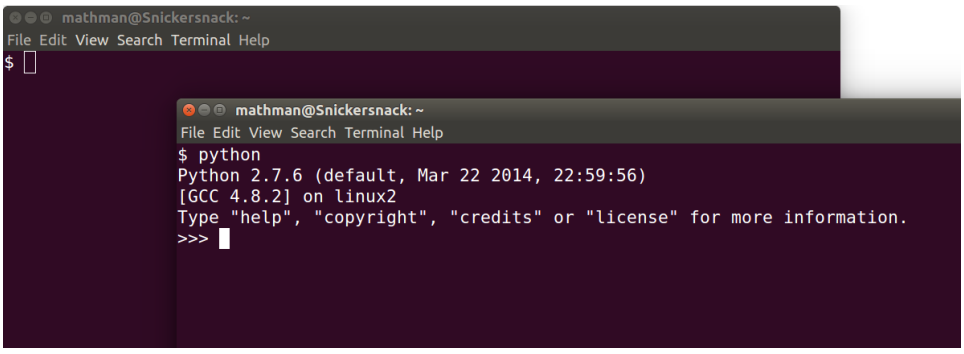


Figure 4.5: A Python shell and a command shell in Ubuntu linux.



Now type the first three lines of the following code into your Python shell. **Do not type in the >>> prompts**. After each line, type the Enter key. A new prompt will appear. 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.

```
>>> x1=2
>>> x2=0.5*(x1+2/x1)
>>> print x2
1.5
```

Now open a text editor like gedit, eclipse, komodo, bbedit, textwrangler, Bluefish, jEdit, Kate, Notepad++, textedit, lightable, bracket, sublime text, ultra edit, geany, scribes, and so forth. Notepad will do but you have to make sure to save you file as a text file. If you are already familiar with them, feel free to use emacs or Vim, but if you don't, leave these for later.

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
```

Save the file as **babyl.py** and verify that the program you are using doesn't try to append a different file type to it like **.txt**. Exit the editor and go to to the command shell (not the python shell). You should see the file you just saved.

In windows, type

```
$ dir babl.py
```

On a mac or in linux, type

```
$ ls babyl.py
```

You just asked the computer to show you a list of all the programs in the local folder (directory) named **babyl.py**. You should see one line, which looks something like this:

```
babyl.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.

When you have found your file, type the following:

```
$ python babyl.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 line.

Now for one more run. Go to the command line again and type

```
$ idle
```

followed by Enter .

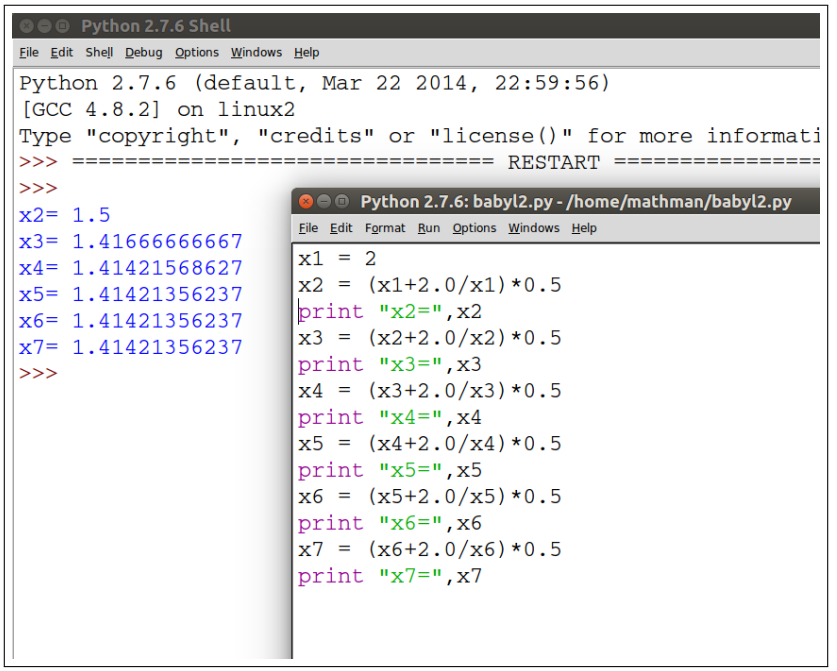
A new and slightly different looking Python shell will open up. You may also be able to find an idle icon on your computer. On the idle menu select **file** > **new** (or equivalently, Ctrl n). A new window will pop open. Copy the entire text of the **babyl.py** program into this window. Then hit the f5 key. You will be prompted for a file name - save the file this time as **babyl2.py**. When you click OK, the program will run in the idle-python window. See figure 4.6.

We've implemented the repeated calculations of the Babylonian algorithm by brute force, namely, by typing the same thing in over and over again. This works fine if you only have to do it a few times. In later chapters we will revise the implementation so that you don't have to type the same thing over and over again.

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 be able to tell which files are Python programs at a glance, you will want to be able to see the file extension, which must be **.py**. To display file extensions, search for the string **file extensions** in the control panel (or just open up a search window). From the Folder Options Menu (figure 4.7), remove the check mark next to the line that says "**Hide file extensions for known file types**" and then click on **Apply**

Figure 4.6: The idle Python window.

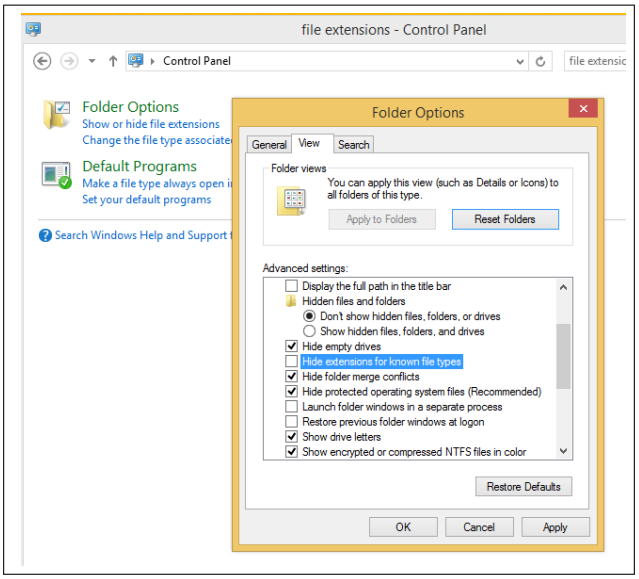


File extensions are normally shown on Mac and Linux systems so no additional modification is required.

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. Explain the difference between command shell and command line. How do 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

Figure 4.7: Show file extensions for removing the check box next to **hide extensions for know file types**, then click **Apply**.



(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 Python shell by typing **`python`** 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.

Chapter 5

iPython Notebooks

IPython notebooks are not really part of Python, and you may wish to skip this chapter entirely. Notebooks provide a way to combine Python code, documentation, analysis, output, and figures all together in one file. They are particularly convenient for student projects. To a certain extent they resemble the type of interface you might see in a commercial program like Mathematica or Matlab.

The key to iPython notebooks is the iPython notebook viewer, which is implemented using a program called **`jupyter`** in your web Browser. Notebooks are stored in a special file format called JSON. That means that you can't edit them or change them except using special software. If you have a web browser on your computer like Firefox, Internet Explorer, or Safari, you can install iPython. Chances are, iPython was automatically installed when you installed Python.

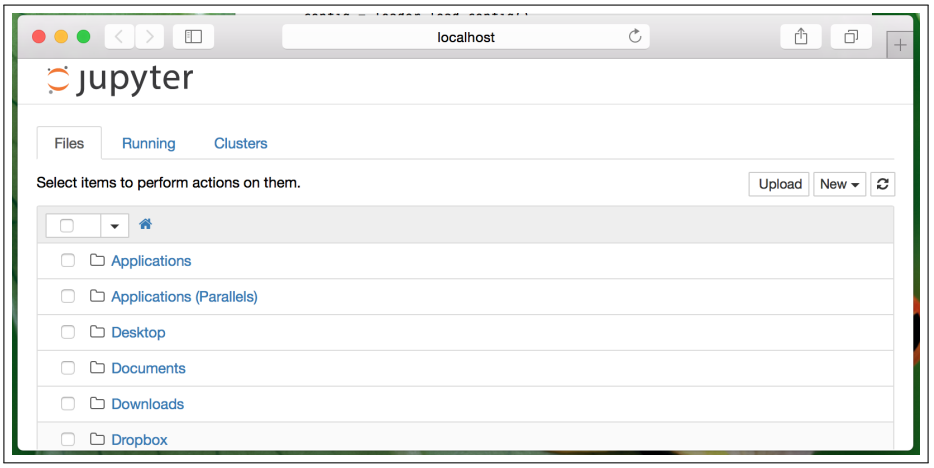


Figure 5.1: The iPython-jupyter iterface on a mac. Open this interface by typing **`ipython notebook`** in a terminal window, such as **`cmd.exe`**. When you first open the interfae all you see is a listing of accessible files and folders. Notebooks will end with the letters **`.ipynb`**.

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[1]

[1]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

```
ipython 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 5.1.

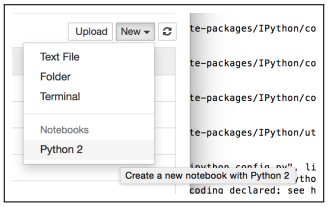


Figure 5.2: Creating a new notebook from the menu.

To create a new notebook, click on the drop-down menu labeled **`New`** and select **`python 2`** (fig. 5.2). Your new notebook appears in a tab labeled **`Untitled`** (fig. 5.3).

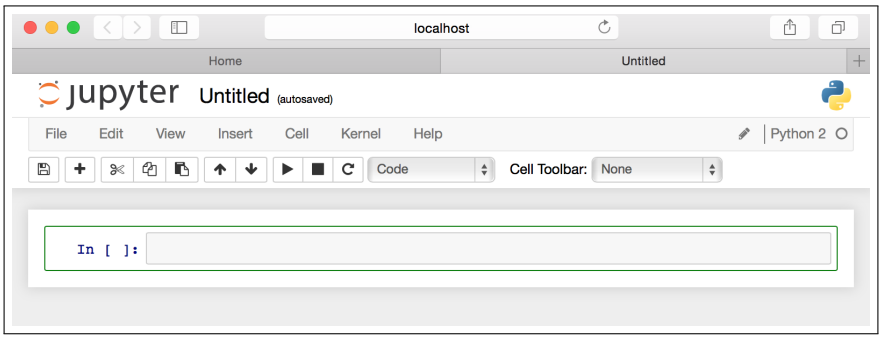


Figure 5.3: A new empty notebook.

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

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!

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.

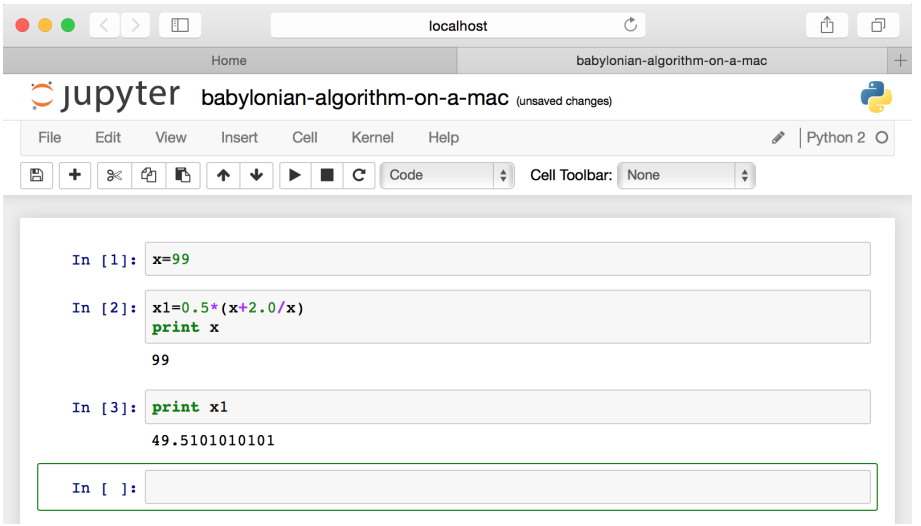


Figure 5.4: One calculation of the Babylonian Algorithm in an iPython notebook.

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 key 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 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 5.4).

Notebooks are particularly useful for combining both text and graphics. Instead of using a **show()** command (chapter 22) all graphics output can be redirected to the notebook and the corresponding **show()** call omitted if you include the command

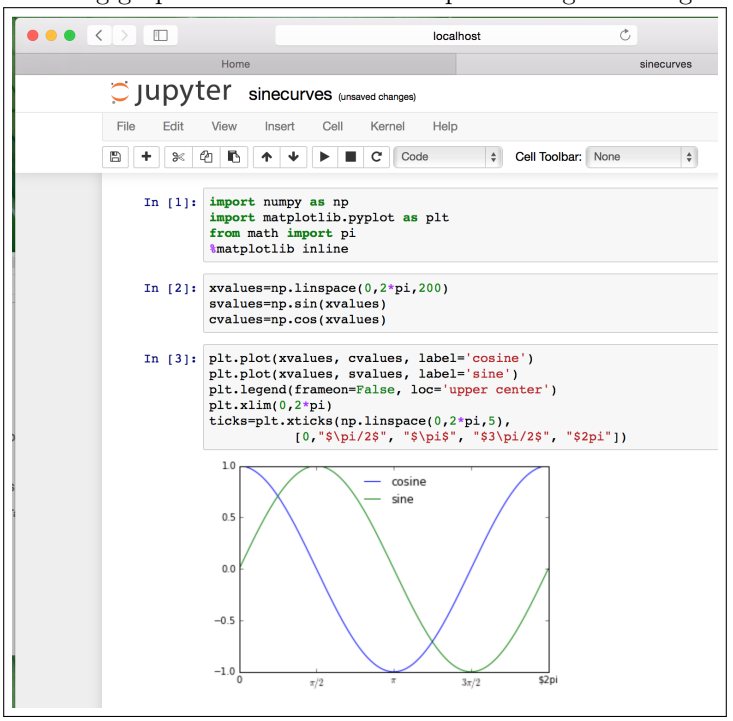
```
%matplotlib inline
```

before your first graphics command (fig. 5.5).

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.

Figure 5.5: Including graphics in the notebook requires using the "magic" command.



Exercises

1. Create a new ipython notebook on your system.
2. Inside your notebook create a cell that defines a function **`f(x,a)`** to do one update of the Babylonian algorithm for $\sqrt{a}$.
3. Write a program, in a different cell, that calculates $\sqrt{42}$ using the function you created in exercise 2.
4. Go back to the top of your notebook and insert a new cell. Write the line **`import math`**, and then execute the cell
5. 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 Baybylonian algorithm. Create a new Markdown cell and type in your observations.

Chapter 6

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

$$117_{10} = (1 \times 10^2) + (1 \times 10^1) + (7 \times 10^0) \tag{6.1}$$

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

$$117 = 64 + 32 + 16 + 4 + 1 \tag{6.2}$$

$$= (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \tag{6.3}$$

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

$$117_{10} = 0111\ 0101_2 \tag{6.4}$$

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

$$117_{10} = (7 \times 16^1) + (5 \times 16^0) \tag{6.5}$$

we can write

$$117_{10} = 75_{16} \tag{6.6}$$

We can also convert 4 bit grouping individually to hexadecimal:

$$117_{10} = \underbrace{0111}_{7}\ \underbrace{0101}_{5} = 75_{16} \tag{6.7}$$

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

$$117_{10} = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\ \ \boxed{0}\boxed{1}\boxed{0}\boxed{1} \tag{6.8}$$

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 6.1. 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:

$$1.5 \text{ terabytes } \times \frac{(1000)^4 \text{ bytes/terabyte}}{(1024)^4 \text{ bytes/tebabyte}} = 1.36424 \text{ tebabytes} \tag{6.9}$$

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

$$0000_2 = 0_{10},\ 0001_2 = 1_{10},\ 0010_2 = 2_{10}, \ldots, 1110_2 = 14_{10},\ 1111_2 = 15_{10} \tag{6.10}$$

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

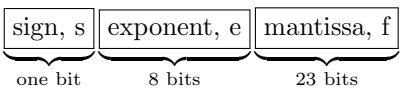
There is no decimal point in a computer word so we cannot represent anything except for integers directly. Instead, we approximate real numbers by their **floating point representation**

$$x = (\pm 1) \times (m) \times 10^E \tag{6.11}$$

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

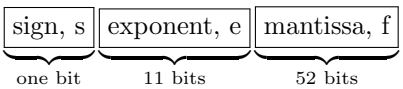
There are three standard ways of doing this, depending upon whether one, two or four words of 32 bit memory are used. Understanding these implementations will help you to understand why computations work the way they do.

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.