Introduction to Python Programming

David Read
Christine Reilly

Python 3.8

Version: 01.02.02

Cover art:

Navy admiral Grace Hopper. Creative Commons Attribution
`https://tophat.com/blog/grace-hopper-yale-computer-science/`

Circuit board. Creative Commons Zero license
`https://www.needpix.com/photo/download/1800090/pcb-printed-circuit-board-components-electronic-resistor-transistor-capacitor`

Python Logo. www.python.org, GPL via Wikimedia Commons
`http://www.gnu.org/licenses/gpl.html`

Printing history:

January 2023: Added a new chapter "Exploring Computer Science" to provide students with insights into the field of computer science.

June 2021: Added sample text from the mbox-small.txt file to chapters 9 (lists) and 12 (regular expressions) to clarify the output dsplayed by the sample code. Thank you to students Emily Hey and Sloane Zwanger for suggesting this update.

May 2021: Fixed minor errata found during the spring 2021 semester. Updated string formatting to use more recent {} syntax (See: `https://docs.python.org/3/library/string.html#string.Formatter`)

January 2021: Using chapters 1-16 of *Python for Informatics: Exploring Information* and material from chapters 1-8 of *Introduction to Computer Science and Programming, Java Edition* to create a textbook for a 100-level Python introductory course.

July 2019: Cleanup of Methods chapter, including moving OO instances discussion into a separate appendix.

January 2019: Minor clarifications and cleanup throughout the chapters.

July 2018: Minor text cleanup in chapter 1 (variables). Clarified the exercise description in chapter 6. Modified Pi Lab 7 to simplify the wiring (reducing from 8 LEDs to 5).

December 2017: Minor text updates in chapter 6 (Strings) including moving scattered exercises to the end of the chapter. Updated debugging instructions for PiLab operation in Appendix B (Pi Labs)

October 2017: Fixed definite and indefinite loops discussion regarding while and for statements in chapter 5 (iteration)

August 2017: Added section on casting to chapter 1 and updated Raspberry Pi setup instructions based on raspi-config changes

December 2016: Minor typographical updates

September 2016: Example code cleanup

August 2016: Updated and extended the Pi setup documentation

June 2016: Added content specific to Raspberry Pi (appendices for Pi setup and labs)

May 2016: General cleanup and editing

April 2016: Added a new chapter on recursion (chapter 7). Major revisions to chapter 8.

March 2016: Major revisions to chapter 6

February 2016: Major revisions to chapters 4 and 5

January 2016: Major revisions to chapters 0 through 3 beginning the creation of a Java textbook *Introduction to Computer Science and Programming, Java Edition* from *Python for Informatics: Exploring Information* by *Dr. Charles R. Severance*.

December 2015: Editorial pass thanks to Sue Blumenberg.

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth. Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

The LaTeX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from `http://www.thinkpython.com`.

Introduction to Computer Science and Programming: Python Edition

David Read and Dr. Christine Reilly

Version 1.0.0

Preface

Students learn in different ways - a seemingly obvious statement. Reviewing the literature on education, it appears as though the process of learning has been studied and written about for as long as people have desired to pass on knowledge. Various theories and models for learning have waxed and waned.

For example, one 7-style theory[1] separates learners into visual, physical, aural, verbal, logical, social and solitary learners. An alternative style list, focused on senses, is provided by the VARK modalities[2] of visual, aural, read/write and kinesthetic.

Regardless of the specific model chosen, there is basic agreement that *different* people succeed by having *different* ways of interacting with material in order to learn it. Further, any one person tends to benefit by having materials which present the class content using multiple modes.

This book's goal is to support the reading, verbal, and solitary modes of learning about computer science and Python programming. This compliments class presentations (visual, aural, logical, social modes) and hands-on lab work (write, kinesthetic, and physical modes). Put another way, this book does not provide a stand-alone or self-study platform for learning to program a computer.

But why create a book focused on Python programming at all? Python is a very mature programming language. There are many books that seek to teach the language and several of those also introduce programming and computer science concepts. My short answer to "why?" is simply, "because I wanted one that aligned well with my approach to teaching computer science and Python."

This book focuses on introducing the mechanics of programming using a third generation language (3GL), specifically Python. Most of the programming concepts apply to many other 3GLs such as C and Java.

What the book does not expound upon[3] are object-oriented (OO) concepts which are vital to being an effective Python programmer. For my intro class, OO is discussed, reviewed and applied repeatedly in lectures and labs in the later half of

[1] `http://www.learndash.com/7-major-learning-styles-which-one-is-you/`
[2] `http://vark-learn.com/introduction-to-vark/the-vark-modalities/`
[3] but arguably should

the semester. I have punted in terms of attempting to document OO in this book. It remains a future to-do.

The foundation for this book comes from another open-source book. I took an on-line Python course taught by Dr. Charles Severance who was using a book he had written and released under an open license. I really liked his approach to the material and the writing style he used in the book.

I decided that having an open book which I could freely alter and extend would allow me to fill a learning mode gap that my slides, presentations, and lab assignments were unlikely to fill effectively. The result is this book. Combined with lectures, slides and labs, I hope it serves as a useful resource for students taking introductory computer science with me.

Finally, I would be remiss if I did not highlight a point I endeavor to reinforce in class, chiefly, *you cannot learn to program simply by reading about programming or by reviewing completed programs*. Programming a computer is very similar to learning to play an instrument.

Just as you learn to play the piano by repeatedly learning to play new pieces, you learn to program by going through the process of writing new programs. *Repeatedly encountering novel requirements and having to figure out the programming constructs necessary to meet those requirement defines programming.* **Programming is a skill that has to be developed through doing, not observing.**

I wish you all the best as you begin your journey into computer science and programming. Whether you intend to pursue computer science as a major or you are just curious about computers and what they can do, I think you'll enjoy exploring logic and the basic process of creating software.

David S. Read
www.monead.com
Schenectady, NY, USA
August 5, 2016

Python for Informatics: Remixing an Open Book

(Charles R. Severance)

This following text is from the prior version of this book (by Dr. Charles Severance) which was written as an introduction to Python programming.

It is quite natural for academics who are continuously told to "publish or perish" to want to always create something from scratch that is their own fresh creation. This book is an experiment in not starting from scratch, but instead "remixing" the book titled *Think Python: How to Think Like a Computer Scientist* written by Allen B. Downey, Jeff Elkner, and others.

In December of 2009, I was preparing to teach **SI502 - Networked Programming** at the University of Michigan for the fifth semester in a row and decided it was time to write a Python textbook that focused on exploring data instead of understanding algorithms and abstractions. My goal in SI502 is to teach people lifelong data handling skills using Python. Few of my students were planning to be professional computer programmers. Instead, they planned to be librarians, managers, lawyers, biologists, economists, etc., who happened to want to skillfully use technology in their chosen field.

I never seemed to find the perfect data-oriented Python book for my course, so I set out to write just such a book. Luckily at a faculty meeting three weeks before I was about to start my new book from scratch over the holiday break, Dr. Atul Prakash showed me the *Think Python* book which he had used to teach his Python course that semester. It is a well-written Computer Science text with a focus on short, direct explanations and ease of learning.

The overall book structure has been changed to get to doing data analysis problems as quickly as possible and have a series of running examples and exercises about data analysis from the very beginning.

Chapters 2–10 are similar to the *Think Python* book, but there have been major changes. Number-oriented examples and exercises have been replaced with data-oriented exercises. Topics are presented in the order needed to build increasingly sophisticated data analysis solutions. Some topics like `try` and `except` are pulled forward and presented as part of the chapter on conditionals. Functions are given very light treatment until they are needed to handle program complexity rather than introduced as an early lesson in abstraction. Nearly all user-defined functions have been removed from the example code and exercises outside of Chapter 4. The word "recursion"[4] does not appear in the book at all.

In chapters 1 and 11–16, all of the material is brand new, focusing on real-world uses and simple examples of Python for data analysis including regular expressions for searching and parsing, automating tasks on your computer, retrieving data across the network, scraping web pages for data, using web services, parsing

[4]Except, of course, for this line.

XML and JSON data, and creating and using databases using Structured Query Language.

The ultimate goal of all of these changes is a shift from a Computer Science to an Informatics focus is to only include topics into a first technology class that can be useful even if one chooses not to become a professional programmer.

Students who find this book interesting and want to further explore should look at Allen B. Downey's *Think Python* book. Because there is a lot of overlap between the two books, students will quickly pick up skills in the additional areas of technical programming and algorithmic thinking that are covered in *Think Python*. And given that the books have a similar writing style, they should be able to move quickly through *Think Python* with a minimum of effort.

As the copyright holder of *Think Python*, Allen has given me permission to change the book's license on the material from his book that remains in this book from the GNU Free Documentation License to the more recent Creative Commons Attribution — Share Alike license. This follows a general shift in open documentation licenses moving from the GFDL to the CC-BY-SA (e.g., Wikipedia). Using the CC-BY-SA license maintains the book's strong copyleft tradition while making it even more straightforward for new authors to reuse this material as they see fit.

I feel that this book serves an example of why open materials are so important to the future of education, and want to thank Allen B. Downey and Cambridge University Press for their forward-looking decision to make the book available under an open copyright. I hope they are pleased with the results of my efforts and I hope that you the reader are pleased with *our* collective efforts.

I would like to thank Allen B. Downey and Lauren Cowles for their help, patience, and guidance in dealing with and resolving the copyright issues around this book.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Charles Severance is a Clinical Associate Professor at the University of Michigan School of Information.

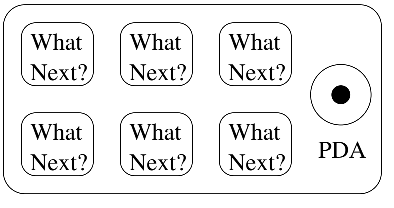
Contents

Chapter 1

Why should you learn to write programs?

Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem. This book assumes that *everyone* needs to know how to program, and that once you know how to program you will figure out what you want to do with your newfound skills.

We are surrounded in our daily lives with computers ranging from laptops to cell phones. We can think of these computers as our "personal assistants" who can take care of many things on our behalf. The hardware in our current-day computers is essentially built to continuously ask us the question, "What would you like me to do next?"



Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite useful and capable of helping us do many different things.

Our computers are fast and have vast amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to "do next". If we knew this language, we could tell the computer to do tasks on our behalf that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

For example, look at the first three paragraphs of this chapter and tell me the most commonly used word and how many times the word is used. While you were able to read and understand the words in a few seconds, counting them is almost painful because it is not the kind of problem that human minds are designed to solve. For a computer the opposite is true, reading and understanding text from a piece of paper is hard for a computer to do but counting the words and telling you how many times the most used word was used is very easy for the computer:

```
python words.py
Enter file:words.txt
to 16
```

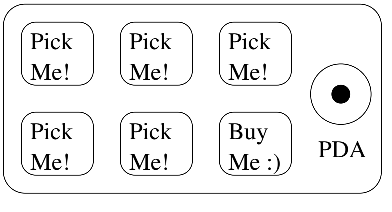
Our "personal information analysis assistant" quickly told us that the word "to" was used sixteen times in the first three paragraphs of this chapter.

This very fact that computers are good at things that humans are not is why you need to become skilled at talking "computer language". Once you learn this new language, you can delegate mundane tasks to your partner (the computer), leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

1.1 Creativity and motivation

This book is designed to introduce people to the concepts inherent in programming a computer. Programming can be a very rewarding job both financially and personally. Building useful, elegant, and clever programs for others to use is a very creative activity. Your computer and smart phone contain many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process. In some situations, when you choose a piece of software, the programmers are directly compensated because of your choice.

If we think of programs as the creative output of groups of programmers, perhaps the following figure is a more sensible version of our PDA:

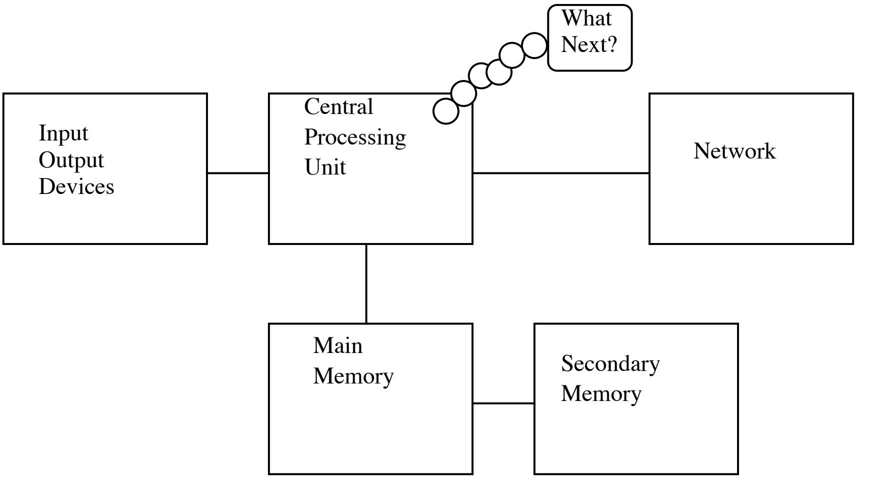


For now, our primary motivation is not to make money or please end users, but instead for us to learn how to write computer programs. We create programs to be more productive and to entertain. When you first start, you will be both the programmer and the end user of your programs. As you gain skill as a programmer and programming feels more creative to you, your thoughts may turn toward developing programs for others.

1.2 Computer hardware architecture

Before we start learning the language we speak to give instructions to computers to develop software, we need to learn a small amount about how computers are built. The architecture used to build most computers, including tablets and smart phones, is based on the work of John von Neumann and is usually referred to as the **von Neumann architecture**.

If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:



The high-level definitions of these parts are as follows:

- The **Central Processing Unit** (or CPU) is the part of the computer that is built to be obsessed with "what is next?" If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask "What next?" three billion times per second. You are going to have to learn how to talk fast to keep up with the CPU.

- The **Main Memory** is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.

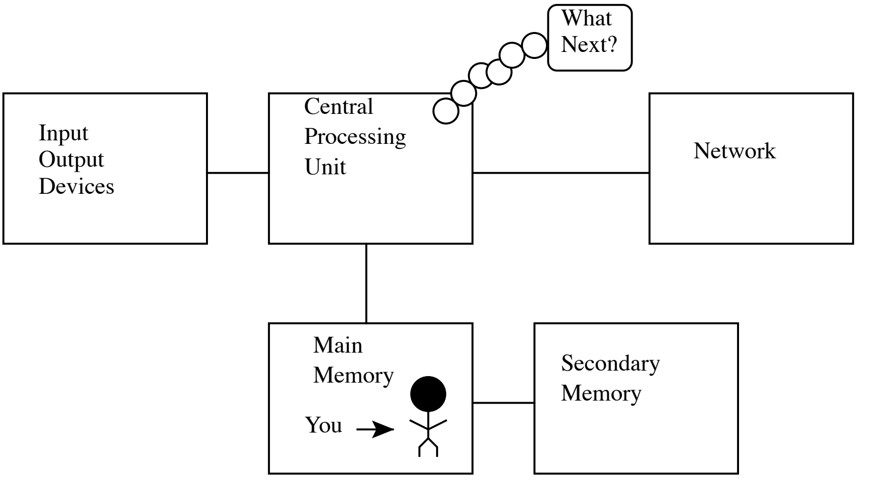
- The **Secondary Memory** is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).

- The **Input and Output Devices** are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer.

- These days, most computers also have a **Network Connection** to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might not always be "up". So in a sense, the network is a slower and at times unreliable form of **Secondary Memory**.

While most of the detail of how these components work is best left to computer builders, it helps to have some terminology so we can talk about these different parts as we write our programs.

As a programmer, your job is to use and orchestrate each of these resources to make the computer carry out the actions you want in order to have your program deliver a result to the user, whether updating a score in a game or analyzing traffic data to find the optimal route home at rush hour. As a programmer you will mostly be "talking" to the CPU and telling it what to do next. Sometimes you will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.



You need to be the person who answers the CPU's "What next?" question. But it would be very uncomfortable to shrink you down to 5mm tall and insert you into the computer just so you could issue a command three billion times per second. So instead, you must write down your instructions in advance. We call these stored instructions a **program**. The act of writing these instructions down and getting the instructions to be correct is called **programming**.

1.3 Understanding programming

In the rest of this book, we will try to turn you into a person who is skilled in the art of programming. In the end you will be a **programmer** — perhaps not a professional programmer, but at least you will have the skills to look at the description of a process (perhaps the rules of a game) and develop a program to automate the process.

In a sense, you need two skills to be a programmer:

- First, you need to know the programming language (Python in this case) - you need to know the vocabulary and the grammar. You need to be able to spell the words in this new language properly and know how to construct well-formed "sentences" in this new language.
- Second, you need to "tell a story". In writing a story, you combine words and sentences to convey an idea to the reader. There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback. In programming, our program is the "story" and the problem you are trying to solve is the "idea".

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as Java or C++. The new programming language has very different vocabulary and grammar but the problem-solving skills will be the same across all programming languages.

You will learn the "vocabulary" and "sentences" of Python pretty quickly. It will take longer for you to be able to write a coherent program to solve a brand-new problem. We teach programming much like we teach writing. We start reading and explaining programs, then we write simple programs, and then we write increasingly complex programs over time. At some point you "get your muse" and see the patterns on your own and can see more naturally how to take a problem and write a program that solves that problem. And once you get to that point, programming becomes a very pleasant and creative process.

We start with the vocabulary and structure of Python programs. Be patient as the simple examples remind you of when you started reading for the first time.

1.4 Words and sentences

Unlike human languages, the Python vocabulary is actually pretty small. We call this "vocabulary" the "reserved words". These are words that have very special meaning to Python. When Python sees these words in a Python program, they have one and only one meaning to Python. Later as you write programs you will make up your own words that have meaning to you called **variables**. You will have great latitude in choosing your names for your variables, but you cannot use any of Python's reserved words as a name for a variable.

When we train a dog, we use special words like "sit", "stay", and "fetch". When you talk to a dog and don't use any of the reserved words, they just look at you with a quizzical look on their face until you say a reserved word. For example, if you say, "Exercise is important. People should sit less and walk more to be healthier", what most dogs likely hear is, "blah blah blah **sit** blah blah blah." That is because "sit" is a reserved word in dog language. Many might suggest that the language between humans and cats has no reserved words[1].

The reserved words in the language where humans talk to Python include the following[2]:

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

That is it, and unlike a dog, Python is already completely trained. When you say "try", Python will try every time you say it without fail.

We will learn these reserved words and how they are used in good time, but for now we will focus on the Python equivalent of "speak" (in human-to-dog language). The nice thing about telling Python to speak is that we can even tell it what to say by giving it a message in quotes:

```
print('Hello world!')
```

And we have even written our first syntactically correct Python sentence. Our sentence starts with the built-in command **print()** which is then given a string of text of our choosing enclosed in single quotes.

If you noticed that "print" is not on the list of reserved words give yourself a pat on the back. Beyond Python's reserved words, there are **functions** and **operators** already built into Python. The "print()" function is used frequently. An example operator is the plus sign (+) which may be used to add numbers. We'll cover these concepts throughout the semester. We'll even learn how to create our own functions!

[1] `http://xkcd.com/231/`
[2] These keywords are for Python 3

1.5 Conversing with Python

Now that we have a word and a simple sentence that we know in Python, we need to know how to start a conversation with Python to test our new language skills.

Before you can converse with Python, you must first install the Python software on your computer and learn how to start Python on your computer. That is too much detail for this chapter so I suggest that you consult https://www.py4e.com/lessons/install for detailed instructions and screencasts of setting up and starting Python on Macintosh and Windows systems. At some point, you will be in a terminal or command window and you will type **python** and the Python interpreter will start executing in interactive mode and appear somewhat as follows:

```
Python 3.7.9 (default, Aug 19 2020, 17:05:11)
[GCC 9.3.1 20200408 (Red Hat 9.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> prompt is the Python interpreter's way of asking you, "What do you want me to do next?" Python is ready to have a conversation with you. All you have to know is how to speak the Python language.

Let's say for example that you did not know even the simplest Python language words or sentences. You might want to use the standard line that astronauts use when they land on a faraway planet and try to speak with the inhabitants of the planet:

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

This is not going so well. Unless you think of something quickly, the inhabitants of the planet are likely to stab you with their spears, put you on a spit, roast you over a fire, and eat you for dinner.

Luckily you brought a copy of this book on your travels, and you thumb to this very page and try again:

```
>>> print('Hello world!')
Hello world!
```

This is looking much better, so you try to communicate some more:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
```

```
Our legend says you will be very tasty with mustard
>>> print('We will have a feast tonight unless you say)
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                     ^
SyntaxError: EOL while scanning string literal
>>>
```

The conversation was going so well for a while and then you made the tiniest mistake using the Python language and Python brought the spears back out.

At this point, you should also realize that while Python is amazingly complex and powerful and very picky about the syntax you use to communicate with it, Python is *not* intelligent. You are really just having a conversation with yourself, but using proper syntax.

In a sense, when you use a program written by someone else the conversation is between you and those other programmers with Python acting as an intermediary. Python is a way for the creators of programs to express how the conversation is supposed to proceed. And in just a few more chapters, you will be one of those programmers using Python to talk to the users of your program.

Before we leave our first conversation with the Python interpreter, you should probably know the proper way to say "good-bye" when interacting with the inhabitants of Planet Python:

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined

>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
             ^
SyntaxError: invalid syntax

>>> quit()
```

You will notice that the error is different for the first two incorrect attempts. The second error is different because **if** is a reserved word and Python saw the reserved word and thought we were trying to say something but got the syntax of the sentence wrong.

The proper way to say "good-bye" to Python is to enter **quit()** at the interactive chevron >>> prompt. It would have probably taken you quite a while to guess that one, so having a book handy probably will turn out to be helpful.

1.6 Terminology: interpreter and compiler

Python is a **high-level** language intended to be relatively straightforward for humans to read and write and for computers to read and process efficiently. Other high-level languages include Java, C++, PHP, Ruby, Basic, Perl, JavaScript, and many more. The actual hardware inside the Central Processing Unit (CPU) does not understand any of these high-level languages.

The CPU understands a language we call **machine language**. Machine language is very simple and frankly very tiresome to write because it is represented all in zeros and ones:

```
001010001110100100101010000001111
11100110000011101010010101101101
...
```

Machine language seems quite simple on the surface, given that there are only zeros and ones, but its syntax is even more complex and far more intricate than Python. So very few programmers ever write machine language. Instead we use various translators to allow programmers to write in high-level languages like Python or Java and these translators convert the programs to machine language for actual execution by the CPU.

Since machine language is tied to the computer hardware, machine language is not **portable** across different types of hardware. Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

These programming language translators fall into two general categories: (1) interpreters and (2) compilers.

An **interpreter** reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.

Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later. We use the term **variable** to refer to the labels we use to refer to this stored data.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

In this example, we ask Python to remember the value six and use the label **x** so we can retrieve the value later. We verify that Python has actually remembered the value using **print**. Then we ask Python to retrieve **x** and multiply it by seven and put the newly computed value in **y**. Then we ask Python to print out the value currently in **y**.

Even though we are typing these commands into Python one line at a time, Python is treating them as an **ordered sequence of statements** with later statements able to retrieve data created in earlier statements. We are writing our first simple paragraph with four sentences in a logical and meaningful order.

It is the nature of an **interpreter** to be able to have an interactive conversation as shown above. A **compiler** needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.

If you have a Windows system, often these executable machine language programs have a suffix of ".exe" or ".dll" which stand for "executable" and "dynamic link library" respectively. In Linux and Macintosh, there is no suffix that uniquely marks a file as executable.

If you were to open an executable file in a text editor, it would look completely crazy and be unreadable:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

It is not easy to read or write machine language, so it is nice that we have **interpreters** and **compilers** that allow us to write in high-level languages like Python or C.

Now at this point in our discussion of compilers and interpreters, you should be wondering a bit about the Python interpreter itself. What language is it written in? Is it written in a compiled language? When we type "python", what exactly is happening?

The Python interpreter is written in a high-level language called "C". You can look at the actual source code for the Python interpreter by going to `www.python.org` and working your way to their source code. So Python is a program itself and it is compiled into machine code. When you installed Python on your computer (or the vendor installed it), you copied a machine-code copy of the translated Python program onto your system. In Windows, the executable machine code for Python itself is likely in a file with a name like:

```
C:\Python27\python.exe
```

That is more than you really need to know to be a Python programmer, but sometimes it pays to answer those little nagging questions right at the beginning.

One other question might have occurred to you, "why are some languages interpreted and other compiled?" There are a variety of reasons but one key difference is speed. A program written using an interpreted language will generally run more slowly on a given computer than the same program written in a compiled language. This is because the compilation step produces a file that is specifically designed to be run by the computer, meaning it has less work to do when it runs the compiled program.

On the flip side, an interpreted language has the advantage of allowing us to experiment one statement at a time to get a program working. We can test our code as we type it in rather than waiting to complete an entire program so that it can be compiled.

1.7 Writing a program

Typing commands into the Python interpreter is a great way to experiment with Python's features, but it is not recommended for solving more complex problems.

When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a **script**. By convention, Python scripts have names that end with `.py`.

To execute the script, you have to tell the Python interpreter the name of the file. In a Unix or Windows command window, you would type `python hello.py` as follows:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
$
```

The "$" is the operating system prompt, and the "cat hello.py" is showing us that the file "hello.py" has a one-line Python program to print a string.

We call the Python interpreter and tell it to read its source code from the file "hello.py" instead of prompting us for lines of Python code interactively.

You will notice that there was no need to have **quit()** at the end of the Python program in the file. When Python is reading your source code from a file, it knows to stop when it reaches the end of the file.

1.8 What is a program?

The definition of a **program** at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple **hello.py** script is a program. It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program.

It might be easiest to understand what a program is by thinking about a problem that a program might be built to solve, and then looking at a program that would solve that problem.

Lets say you are doing Social Computing research on Facebook posts and you are interested in the most frequently used word in a series of posts. You could print out the stream of Facebook posts and pore over the text looking for the most common word, but that would take a long time and be very mistake prone. You would be smart to write a Python program to handle the task quickly and accurately so you can spend the weekend doing something fun.

For example, look at the following text about a clown and a car. Look at the text and figure out the most common word and how many times it occurs.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Then imagine that you are doing this task looking at millions of lines of text. Frankly it would be quicker for you to learn Python and write a Python program to count the words than it would be to manually scan the words.

The even better news is that I already came up with a simple program to find the most common word in a text file. I wrote it, tested it, and now I am giving it to you to use so you can save some time.

```
name = input('Enter file: ')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
    counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

You don't even need to know Python to use this program. You will need to get through Chapter 10 of this book to fully understand the awesome Python techniques that were used to make the program. You are the end user, you simply use the program and marvel at its cleverness and how it saved you so much manual effort. You simply type the code into a file called **words.py** and run it or you download the source code from `https://www.py4e.com/code3/` and run it.

This is a good example of how Python and the Python language are acting as an intermediary between you (the end user) and me (the programmer). Python is a way for us to exchange useful instruction sequences (i.e., programs) in a common language that can be used by anyone who installs Python on their computer. So neither of us are talking *to Python*, instead we are communicating with each other *through* Python.

1.9 The building blocks of programs

In the next few chapters, we will learn more about the vocabulary, sentence structure, paragraph structure, and story structure of Python. We will learn about the powerful capabilities of Python and how to compose those capabilities together to create useful programs.

There are some low-level conceptual patterns that we use to construct programs. These constructs are not just for Python programs, they are part of every programming language from machine language up to the high-level languages.

input: Get data from the "outside world". This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.

output: Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.

sequential execution: Perform statements one after another in the order they are encountered in the program.

conditional execution: Check for certain conditions and then execute or skip a sequence of statements.

repeated execution: Perform some set of statements repeatedly, usually with some variation.

reuse: Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

It sounds almost too simple to be true, and of course it is never so simple. It is like saying that walking is simply "putting one foot in front of the other". The "art" of writing a program is composing and weaving these basic elements together many times over to produce something that is useful to its users.

The word counting program above directly uses all of these patterns except for one. Any idea which one?

1.10 What could possibly go wrong?

As we saw in our earliest conversations with Python, we must communicate very precisely when we write Python code. The smallest deviation or mistake will cause Python to give up looking at your program.

Beginning programmers often take the fact that Python leaves no room for errors as evidence that Python is mean, hateful, and cruel. While Python seems to like everyone else, Python knows them personally and holds a grudge against them. Because of this grudge, Python takes our perfectly written programs and rejects them as “unfit” just to torment us.

There is little to be gained by arguing with Python. It is just a tool. It has no emotions and it is happy and ready to serve you whenever you need it. Its error messages sound harsh, but they are just Python’s call for help. It has looked at what you typed, and it simply cannot understand what you have entered.

Python is much more like a dog, loving you unconditionally, having a few key words that it understands, looking at you with a sweet look on its face and waiting for you to say something it understands. When Python says something like:

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt 'Hello world'
  File "<stdin>", line 1
    primt 'Hello world'
                      ^
SyntaxError: invalid syntax
>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
              ^
SyntaxError: invalid syntax
>>>
```

it is simply wagging its tail and saying, “You seemed to say something but I just don’t understand what you meant.”

As your programs become increasingly sophisticated, you will encounter three general types of errors:

Syntax errors: These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the “grammar” rules of Python. Python does its best to point right at the line and character where it noticed it was confused. The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python *noticed* it was confused. So the line and character that Python indicates in a syntax error may just be a starting point for your investigation.

Logic errors: A logic error occurs when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another. Logic errors do not prevent the program from compiling. A good example of a logic error might be, "take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle."

Semantic errors: A semantic error occurs when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you *intended* for it to do. A simple example would be if you were giving a person directions to a restaurant and said, "...when you reach the intersection with the gas station, turn left and go one mile and the restaurant is a red building on your left." Your friend is very late and calls you to tell you that they are on a farm and walking around behind a barn, with no sign of a restaurant. Then you say "did you turn left or right at the gas station?" and they say, "I followed your directions perfectly, I have them written down, it says turn left and go one mile at the gas station." Then you say, "I am very sorry, because while my instructions were syntactically correct, they sadly contained a small but undetected semantic error.". The issue in this case being that you meant to tell your friend to turn right instead of left. The order of steps is correct (logically) but the semantics are incorrect (right versus left). Logic and semantic errors are very related so don't get hung up on trying to differentiate between them at this point.

Again in all three types of errors, Python is merely trying its hardest to do exactly what you have asked.

1.11 Debugging

When Python spits out an error or even when it gives you a result that is different from what you had intended, then begins the hunt for the cause of the error. Debugging is the process of finding the cause of the error in your code. When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

Ruminating: Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even to yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

1.12 The learning journey

As you progress through the rest of the book, don't be afraid if the concepts don't seem to fit together well the first time. When you were learning to speak, it was not a problem for your first few years that you just made cute gurgling noises. And it was OK if it took six months for you to move from simple vocabulary to simple sentences and took 5-6 more years to move from sentences to paragraphs, and a few more years to be able to write an interesting complete short story on your own.

We want you to learn Python much more rapidly, so we teach it all at the same time over the next few chapters. But it is like learning a new language that takes time to absorb and understand before it feels natural. That leads to some confusion as we visit and revisit topics to try to get you to see the big picture while we are defining the tiny fragments that make up that big picture. While the book is written linearly, and if you are taking a course it will progress in a linear fashion, don't hesitate to be very nonlinear in how you approach the material. Look forwards and backwards and read with a light touch. By skimming more advanced material without fully understanding the details, you can get a better understanding of the "why?" of programming. By reviewing previous material and even redoing earlier exercises, you will realize that you actually learned a lot of material even if the material you are currently staring at seems a bit impenetrable.

Usually when you are learning your first programming language, there are a few wonderful "Ah Hah!" moments where you can look up from pounding away at some rock with a hammer and chisel and step away and see that you are indeed building a beautiful sculpture.

If something seems particularly hard, there is usually no value in staying up all night and staring at it. Take a break, take a nap, have a snack, explain what you are having a problem with to someone (or perhaps your dog), and then come back to it with fresh eyes. I assure you that once you learn the programming concepts in the book you will look back and see that it was all really easy and elegant and it simply took you a bit of time to absorb it.

1.13 Glossary

bug: An error in a program.

central processing unit: The heart of any computer. It is what runs the software that we write; also called "CPU" or "the processor".

compile: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

high-level language: A programming language like Python that is designed to be easy for humans to read and write.

interactive mode: A way of using the Python interpreter by typing commands and expressions at the prompt.

interpret: To execute a program in a high-level language by translating it one line at a time.

low-level language: A programming language that is designed to be easy for a computer to execute; also called "machine code" or "assembly language".

machine code: The lowest-level language for software, which is the language that is directly executed by the central processing unit (CPU).

main memory: Stores programs and data. Main memory loses its information when the power is turned off.

parse: To examine a program and analyze the syntactic structure.

portability: A property of a program that can run on more than one kind of computer.

print statement: An instruction that causes the Python interpreter to display a value on the screen.

problem solving: The process of formulating a problem, finding a solution, and expressing the solution.

program: A set of instructions that specifies a computation.

prompt: When a program displays a message and pauses for the user to type some input to the program.

secondary memory: Stores programs and data and retains its information even when the power is turned off. Generally slower than main memory. Examples of secondary memory include disk drives and flash memory in USB sticks.

semantics: The meaning of a program.

semantic error: An error in a program that makes it do something other than what the programmer intended.

source code: A program in a high-level language.

1.14 Exercises

Exercise 1.1 What is the function of the secondary memory in a computer?

a) Execute all of the computation and logic of the program
b) Retrieve web pages over the Internet
c) Store information for the long term – even beyond a power cycle
d) Take input from the user

Exercise 1.2 What is a program?

Exercise 1.3 What is the difference between a compiler and an interpreter?

Exercise 1.4 Which of the following contains "machine code"?

a) The Python interpreter
b) The keyboard
c) Python source file
d) A word processing document

Exercise 1.5 What is wrong with the following code:

```
>>> primt('Hello world!')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined
>>>
```

Exercise 1.6 What is wrong with the following code:

```
>>> print('Love the silly walks)
 File "<stdin>", line 1
   print('Love the silly walks)
                              ^
SyntaxError: EOL while scanning string literal
```

Exercise 1.7 What is wrong with the following code:

```
>>> print(I wish pluto was still a full-fledged planet)
 File "<stdin>", line 1
   print(I wish pluto was still a full-fledged planet)
           ^
SyntaxError: invalid syntax
```

Exercise 1.8 Where in the computer is a variable such as "X" stored after the following Python line finishes?

```
x = 123
```

a) Central processing unit
b) Main Memory
c) Secondary Memory
d) Input Devices
e) Output Devices

Exercise 1.9 What will the following program print out:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44

c) x + 1
d) Error because x = x + 1 is not possible mathematically

Exercise 1.10 Explain each of the following using an example of a human capability: (1) Central processing unit, (2) Main Memory, (3) Secondary Memory, (4) Input Device, and (5) Output Device. For example, "What is the human equivalent to a Central Processing Unit"?

Exercise 1.11 How do you fix a "Syntax Error"?

Chapter 2

Variables, expressions, and statements

2.1 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. The values we have seen so far are `1`, `2`, and `'Hello, World!'`

These values belong to different **types**: `2` is an integer, and `'Hello, World!'` is a **string**, so called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The `print` statement also works for integers. We use the `python` command to start the interpreter.

```
python
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating point**.

As you may have guessed, **type()** is another function built in to Python. It reports back the type of information it is given. Compare this to the **print()** function we've been using which displays the supplied information on the screen.

```
>>> type(3.2)
<class 'float'>
```

What about values like `'17'` and `'3.2'`? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `1,000,000`. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

An **assignment statement** creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second assigns the integer `17` to `n`; the third assigns the (approximate) value of π to `pi`.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.14159265359
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Remember Python's reserved words? We saw the list in the previous chapter, here it is again for your reference.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

2.4 Statements

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

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

produces the output

```
1
2
```

Note that the assignment statement produces no visible output.

2.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**.

The operators `+`, `-`, `*`, `/`, and `**` perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32   hour-1   hour*60+minute   minute/60   minute//60   5**2   (5+9)*(15-7)
```

There are two division operators. One uses a single slash and the other uses two slashes. See if you can figure out how they differ from these examples:

```
>>> minute = 59
>>> minute / 60
0.9833333333333333
>>> minute // 60
0
```

The value of `minute` is 59, an integer. In the first division, using the single slash operator, Python gives us a `float` result (0.983...). However, when we use the double slash division operator Python gives us an `int` result (0). The difference is that the single slash division operator performs floating point division while the double slash division operator performs integer division. In the latter case, only whole numbers (integers) may be represented in the answer. Any fractional value is truncated (removed). This is known as `floor` division. This means that Python does not round the integer answer to the nearest integer it simply floors the result, throwing away the fractional value.

2.6 Expressions

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 2.1 Type the following statements in the Python interpreter to see what they do:

```
5
x = 5
x + 1
```

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.

- **E**xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.

- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.

- Operators with the same precedence are evaluated from left to right. So the expression `5-3-1` is 1, not 3, because the `5-3` happens first and then `1` is subtracted from `2`.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

2.8 Modulus operator

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (`%`). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

You can also extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly, `x % 100` yields the last two digits.

2.9 String operations

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs **concatenation**, which means joining the strings by linking them end to end. For example:

```
>>> first = 10
>>> second = 15
>>> print(first + second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

The first output is `25` which is the mathematical sum of 10 and 15. The second output is `100150` which is the concatenation of '100' and '150'.

Concatenation is very useful as a way to build up messages shown to a user or written to a file. We'll use this technique often in our programs to concatenate text labels with information in variables.

2.10 Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `input` that gets input from the keyboard. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string.

```
>>> info = input()
Some silly stuff
>>> print(info)
Some silly stuff
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to `input` to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int` using the `int()` function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
```

We will see how to handle this kind of error later.

2.11 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and in Python they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

2.12 Choosing mnemonic variable names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the **intent** of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[1] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are about 30 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
    print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
    print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, `pizza` and `slice` are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

[1] See `http://en.wikipedia.org/wiki/Mnemonic` for an extended description of the word "mnemonic".

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

```
for word in words:
    print( word )
```

The parts of the code that are defined by Python (`for`, `in`, `print()`, and `:`) are in bold and the programmer-chosen variables (`word` and `words`) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

2.13 Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `US$`, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate $\frac{1}{2\pi}$, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

2.14 Glossary

assignment: A statement that assigns a value to a variable.

concatenate: To join two operands end to end.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

evaluate: To simplify an expression by performing the operations in order to yield a single value.

expression: A combination of variables, operators, and values that represents a single result value.

floating point: A type that represents numbers with fractional parts.

floor division: The operation that divides two numbers and chops off the fractional part.

integer: A type that represents whole numbers.

keyword: A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

mnemonic: A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator: An operator, denoted with a percent sign (`%`), that works on integers and yields the remainder when one number is divided by another.

operand: One of the values on which an operator operates.

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence: The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

string: A type that represents sequences of characters.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

value: One of the basic units of data, like a number or string, that a program manipulates.

variable: A name that refers to a value.

2.15 Exercises

Exercise 2.2 Write a program that uses the `input` function to prompt a user for their name and then welcomes them. Here is what it would look like when run.

```
Enter your name: Chuck
Hello Chuck
```

Exercise 2.3 Write a program to prompt the user for hours and rate per hour to compute gross pay. Note that when run the user will enter the hours and rate. Then the pay will be calculated and printed on the screen.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

We won't worry about making sure our pay has exactly two digits after the decimal place for now. If you want, you can play with the built-in Python `round` function to properly round the resulting pay to two decimal places.

Exercise 2.4 Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width / 2`
2. `width // 2`
3. `height / 3`
4. `1 + 2 * 5`

Use the Python interpreter to check your answers.

Exercise 2.5 Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature. Remember the formula for this conversion is `DegreesF = DegreesC * 9.0 / 5.0 + 32.0`

Chapter 3

Conditional execution

3.1 Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The == operator is one of the **comparison operators**; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

3.2 Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example,

```
x > 0 and x < 10
```

is true only if `x` is greater than 0 *and* less than 10.

`n % 2 == 0 or n % 3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false; that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>> 17 and True
True
```

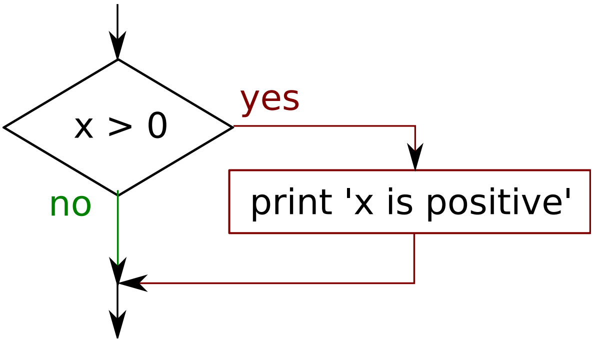
We say that a value is "truthy" or "falsey" depending on how Python treats it when used as a boolean value. So 17 is truthy and 0 is falsey. This flexibility can be useful, but there are some subtleties to it that can be confusing. You might want to avoid it until you are sure you know what you are doing. Further, relying on this conversion may make a program harder to understand.

3.3 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0 :
    print('x is positive')
```

The boolean expression after the `if` statement is called the **condition**. We end the `if` statement with a colon character (:) and the line(s) after the if statement are indented.



If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

`if` statements have the same structure as function definitions or `for` loops[1].The statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called **compound statements** because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a placekeeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0 :
    pass          # need to handle negative values!
```

If you enter an `if` statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

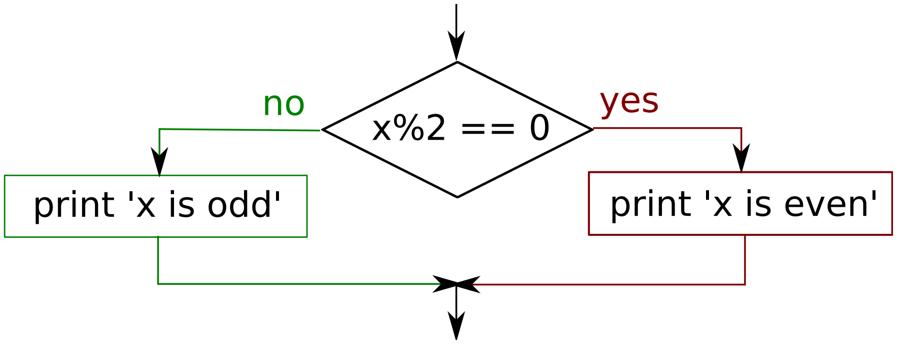
3.4 Alternative execution

A second form of the `if` statement is **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x % 2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.

[1]We will learn about functions in Chapter 4 and loops in Chapter 5.



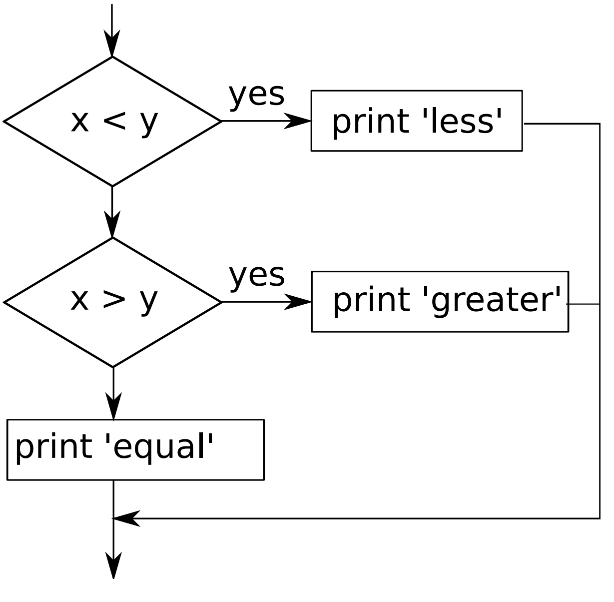
Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

3.5 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` is an abbreviation of "else if." Again, exactly one branch will be executed.



There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

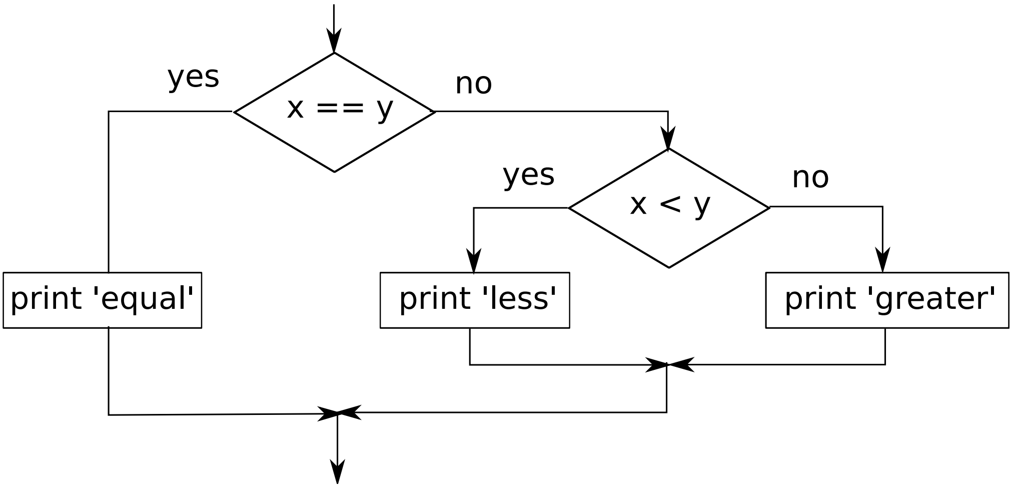
Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

3.6 Nested conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.



Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

3.7 Catching exceptions using try and except

Earlier we saw a code segment where we used the `input` and `int` functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think "oops", and move on to our next statement.

However if you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temperature:fred

Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except". The idea of `try` and `except` is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the `try` and `except` feature in Python as an "insurance policy" on a sequence of statements.

We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('You did not enter a number')
```

Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
You did not enter a number
```

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully. After you learn about looping in the Iteration chapter you can write code that uses a loop to keep asking for user input until the user enters the desired type of data.

3.8 Short-circuit evaluation of logical expressions

When Python is processing a logical expression such as `x >= 2 and (x / y) > 2`, it evaluates the boolean expressions from left to right. Because of the definition of `and`, if `x` is less than 2, the expression `x >= 2` is `False` and so the whole expression is `False` regardless of whether `(x / y) > 2` evaluates to `True` or `False`.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called **short-circuiting** the evaluation.

While this may seem like a fine point, the short-circuit behavior leads to a clever technique called the **guardian pattern**. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x / y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x / y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x / y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

The third calculation failed because Python was evaluating `(x/y)` and `y` was zero, which causes a runtime error. But the first and the second examples did *not* fail because the first part of these expressions `x >= 2` evaluated to `False` so the `(x/y)` was not ever executed due to the **short-circuit** rule and there was no error.

We can construct the logical expression to strategically place a **guard** evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x / y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x / y) > 2
False
>>> x >= 2 and (x / y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

In the first logical expression, `x >= 2` is `False` so the evaluation stops at the `and`. In the second logical expression, `x >= 2` is `True` but `y != 0` is `False` so we never reach `(x/y)`.

In the third logical expression, the `y != 0` is *after* the `(x/y)` calculation so the expression fails with an error.

In the second expression, we say that `y != 0` acts as a **guard** to insure that we only execute `(x/y)` if `y` is non-zero.

3.9 Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them when reading.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

But when you run it, you get an error message:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, because integer division is being used. The solution is to use floating point division (with the single slash).

In general, error messages tell you where the problem was discovered, but that may not be where it was caused.

3.10 Glossary

body: The sequence of statements within a compound statement.

boolean expression: An expression whose value is either `True` or `False`.

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

comparison operator: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch is executed.

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

guardian pattern: Where we construct a logical expression with additional comparisons to take advantage of the short-circuit behavior.

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

traceback: A list of the functions that are executing, printed when an exception occurs.

short circuit: When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

3.11 Exercises

Exercise 3.1 Rewrite your pay computation (exercise 2.3, page 32) to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 3.2 Rewrite your pay program, from above, using `try` and `except` so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input
```

Exercise 3.3 Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
Score   Grade
>= 0.9    A
>= 0.8    B
>= 0.7    C
>= 0.6    D
< 0.6     F

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Run the program repeatedly as shown above to test the various different values for input.

Chapter 4

Functions

4.1 Function calls

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a **function call**:

```
>>> type(32)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type` function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

4.2 Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

These functions are not limited to looking at strings. They can operate on any set of values, as we will see in later chapters.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name). Creating variables whose names match a function will lead to errors when you try to use those functions.

4.3 Type conversion functions

Python also provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use **algorithms** that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

Exercise 4.1 Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

The `random` function is only one of many functions that handle random numbers. The function `randint` takes the parameters `low` and `high`, and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence[1] at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

4.5 Math functions

Python has a `math` module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' from '/usr/local/lib/python3.8/lib-dynload/math.cpython-38-x86_64-linux-gnu.so'>
```

The module object contains the functions and variables defined in the module[2]. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

[1] We'll cover sequences in a future chapter
[2] We'll learn how to create our own modules later in the course

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

4.6 Creating functions

So far, we have only been using the functions that come with Python, but it is also possible to create our own functions. This allows us to add new functions beyond those provided by Python.

A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

The value of `print_lyrics` is a **function object**, which has type `'function'`.

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

4.7 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 4.2 Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 4.3 Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

4.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

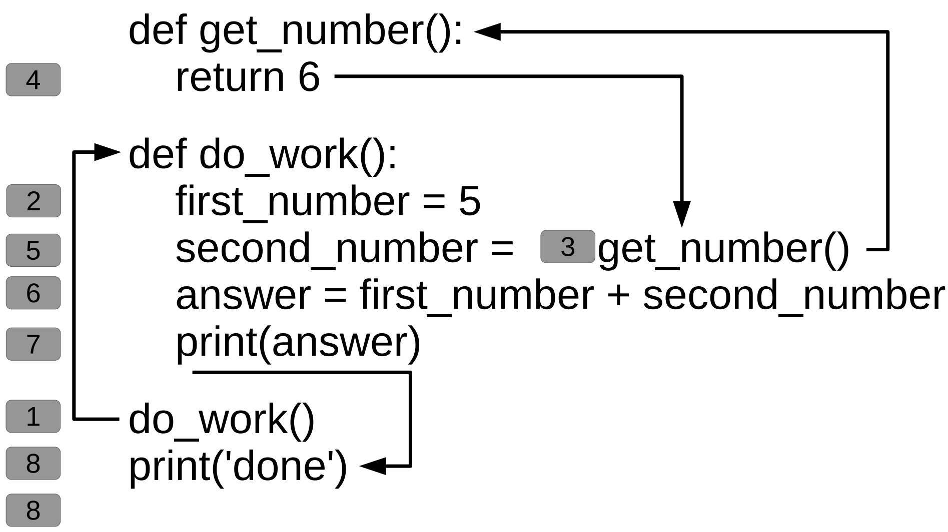
Function *definitions* do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

Here is a depiction of a program, traceflow.py, numbering the statements in the order they will execute. The arrows depict the function calls which cause the flow of execution to change.



Running the `traceflow.py` program would produce this output on the screen:

```
11
done
```

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.9 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979 
```

But in a script, **if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!**

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

```
>>> type(None)
<class 'NoneType'>
```

To return a result from a function, we use the `return` statement in our function. For example, we could make a very simple function called `addtwo` that adds two numbers together and returns a result.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

When this script executes, the `print` statement will print out "8" because the `addtwo` function was called with 3 and 5 as arguments. Within the function, the parameters `a` and `b` were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added`. Then it used the `return` statement to send the computed value back to the calling code as the function result, which was assigned to the variable `x` and printed out.

4.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as "find the smallest value in a list of values". Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

4.12 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print('hello')` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

4.13 Glossary

algorithm: A general process for solving a category of problems.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body: The sequence of statements inside a function definition.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution: The order in which statements are executed during a program run.

fruitful function: A function that returns a value.

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call: A statement that executes a function. It consists of the function name followed by an argument list.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

header: The first line of a function definition.

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an `import` statement that provides access to the data and code defined in a module.

parameter: A name used inside a function to refer to the value passed as an argument.

pseudorandom: Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function: A function that does not return a value.

4.14 Exercises

Exercise 4.4 What is the purpose of the ”def” keyword in Python?

a) It is slang that means ”the following code is really cool”
b) It indicates the start of a function
c) It indicates that the following indented section of code is to be stored for later
d) b and c are both true
e) None of the above

Exercise 4.5 What will the following Python program print out?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Exercise 4.6 Rewrite your pay computation (exercise 3.2, page 43) with time-and-a-half for overtime and create a function called `computepay` which takes two parameters (`hours` and `rate`).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 4.7 Rewrite the grade program (exercise 3.3, page 43) using a function called `computegrade` that takes a score as its parameter and returns a grade as a string.

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F

Program Execution:

Enter score: 0.95
```

```
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Run the program repeatedly to test the various different values for input.

Exercise 4.8 Retype the traceflow program (page 52) into a file on your computer called traceflow.py and run it. Verify you get the expected results.

Next, modify the `def_number()` function to accept a number from the user (use the `input()` function) and return the value entered by the user. Test your updated program and verify the new output is what you expect.

Chapter 5

Iteration

5.1 Updating variables

A common pattern in assignment statements is an assignment statement that updates a variable – where the new value of the variable depends on the old.

```
x = x + 1
```

This means "get the current value of `x`, add 1, and then update `x` with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

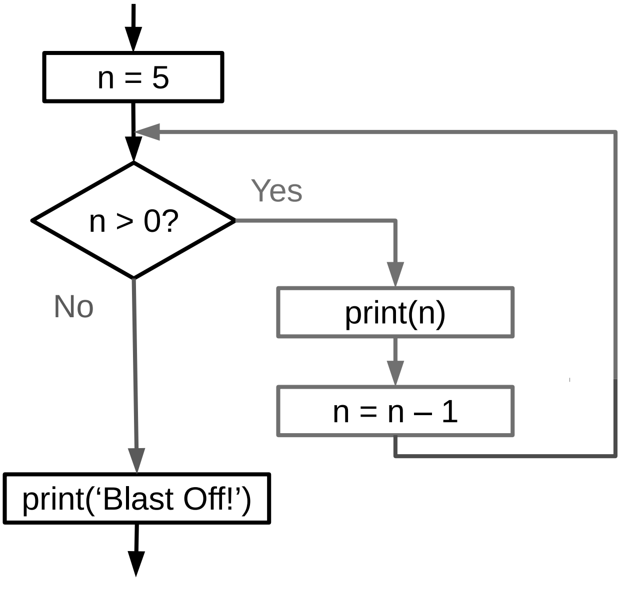
5.2 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the `while` statement. Here is a simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than 0, display the value of `n` and then reduce the value of `n` by 1. When you get to 0, exit the `while` statement and display the word `Blastoff!`"



More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. We call each time we execute the body of the loop an **iteration**. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the **iteration variable**. If there is no iteration variable, the loop will repeat forever, resulting in an **infinite loop**.

5.3 Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no **iteration variable** telling you how many times to execute the loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

5.4 "Infinite loops" and `break`

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the `break` statement to jump out of the loop.

This loop is obviously an **infinite loop** because the logical expression on the `while` statement is simply the logical constant `True`:

```
n = 10
while True:
    print(n)
    n = n - 1
print('Done!')
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value `True`.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using `break` when we have reached the exit condition.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> hello there
hello there
> finished
finished
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

5.5 Finishing iterations with `continue`

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

Here is a sample run of this new program with `continue` added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the `continue` is executed, it ends the current iteration and jumps back to the `while` statement to start the next iteration, thus skipping the `print` statement.

5.6 A word of caution

There is debate among programmers about the use of the `break` and `continue` statements. Some programmers consider these statements as useful for writing concise code. Other programmers dislike using these statements and think that they lead to code that is difficult to read and prone to bugs. A general guideline is to use the `break` and `continue` statements in loops that are short and simple, and to use other approaches for loop control in longer or more complex loops. Your goal is to write code that is easy for others to quickly read and understand.

An additional iteration variable is an alternative to using a `break` statement. An `if` statement can be used in place of a `continue` statement. For example, here is the code from above rewritten without using `break` or `continue`.

```
line = input('> ')
while line != 'done':
    if line[0] != '#':
        print(line)
    line = input('> ')
print('Done!')
```

5.7 Definite loops using `for`

Sometimes we want to loop through a **collection** of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a *definite* loop using a `for` statement. We call the `while` statement an *indefinite* loop because it simply loops until some condition becomes `False`, whereas the `for` loop is looping through a known collection of items so it runs through as many iterations as there are items in the collection.

The syntax of a `for` loop is similar to the `while` loop in that there is a `for` statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable `friends` is a list[1] of three strings and the `for` loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

[1] We will examine lists in more detail in a later chapter.

Translating this `for` loop to English is not as direct as the `while`, but if you think of friends as a **collection**, it goes like this: "Run the statements in the body of the for loop once for each friend *in* the collection named `friends`."

Looking at the `for` loop, **for** and **in** are reserved Python keywords, and `friend` and `friends` are variables.

```
for friend in friends:
    print( 'Happy New Year', friend )
```

In particular, `friend` is the **iteration variable** for the `for` loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The **iteration variable** steps successively through the three strings stored in the `friends` variable.

5.8 Loop patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

5.8.1 Counting and summing loops

For example, to count the number of items in a list, we would write the following `for` loop:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

We initialize the variable `count` to zero before the loop starts, then we write a `for` loop to run through the list of numbers. Our **iteration** variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen "so far".

Once the loop completes, the value of `count` is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a collection of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

In this loop we *do* use the **iteration variable**. Instead of simply adding one to the `count` as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable `total`, it contains the "running total of the values so far". So before the loop starts `total` is zero because we have not yet seen any values, during the loop `total` is the running total, and at the end of the loop `total` is the overall total of all the values in the list.

As the loop executes, `total` *accumulates* the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

5.8.2 Maximum and minimum loops

To find the largest value in a collection such as a list, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
```

```
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the "largest value we have seen so far". Before the loop, we initialize `largest` to the constant `None`. `None` is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `itervar` is 3, since `largest` is `None`, we immediately update `largest` to be 3.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `itervar > largest` triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Again, `smallest` is the "smallest so far" before, during, and after the loop executes. When the loop has completed, `smallest` contains the minimum value in the list.

Again as in counting and summing, the built-in functions `max()` and `min()` make writing these exact loops unnecessary.

The following is a simple version of the Python built-in `min()` function:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

In the function version of the smallest code, we removed all of the `print` statements so as to be equivalent to the `min` function which is already built in to Python.

5.9 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

5.10 Glossary

accumulator: A variable used in a loop to add up or accumulate a result.

counter: A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.

decrement: An update that decreases the value of a variable.

initialize: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

infinite loop: A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

iteration: Repeated execution of a group of statements using either a function that calls itself or a loop.

5.11 Exercises

Exercise 5.1 Write a program which repeatedly reads numbers until the user enters "done". Once "done" is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using `try` and `except` and print an error message and skip to the next number.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.33333333333
```

Exercise 5.2 Modify the above program to first ask the user whether they want the sum, count, mean, maximum, or minimum of the numbers they enter. Then allow the user to enter the numbers, handling errors as above, and using "done“ to signal they are finished. Finally, show the desired output based on their initial choice (sum, count, mean, max or min). Be sure to label the value so it is clear what it represents. Here is an example run:

```
Do you want the sum, count, mean, min, or max? min
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
Minimum is 4
```

Chapter 6

Recursion

6.1 Divide and conquer

One approach that we may use when faced with a large task is to break it into smaller tasks, each a subset of the original large one. We work out each smaller task and in the process end up completing the larger one.

This process of expressing an algorithm by having it apply the same logic to a subset of the data is known as **recursion**.

Here are a couple of visual depictions of recursion.

In this first image we see traditional *Russian dolls*, where a set of dolls, each essentially a larger version of the previous, nest together to form the whole.[1]

[1] By Photo: RK812, Doll carved by Zvezdochkin, painted by Malyutin - Sergiev Posad Museum of Toys, Russia, Public Domain, https://commons.wikimedia.org/w/index.php?curid=5051554

In our second image, we see an artist painting a picture which shows an artist painting a smaller picture, which shows an artist painting a smaller picture...[2]

Both of these examples may help you understand recursion, however the images seem to be repeating the same information at different sizes. When we talk about using recursion in a computer program, it is the size of the data (meaning the values or the number of elements in an list) that is changing. In other words we are working with subsets, not duplicate information.

Here is an example of using recursion in the real world.[3]

> The person behind you in a movie theater asks you what row you're sitting in. You don't want to count, so you ask the person in front of you what row they are sitting in, knowing that you will respond to the person behind you one greater than the answer the person in front of you gives. The person in front will ask the person in front of them. This will keep happening until the question reaches the front row, and it is easy to respond: "I'm in row 1!" From there, the correct message (incremented by one by the person in each row) will eventually make its way back to the person who asked.

[2] Stack Exchange Network, http://programmers.stackexchange.com/questions/93826/how-do-i-explain-recursion-to-a-8-years-old-kid

[3] Quora, https://www.quora.com/How-should-I-explain-recursion-to-a-4-year-old

This exemplifies several key aspects of using recursion:

1. The question ("what row am I in?") needs to be rephrased recursively as: "how many people are in front of me + 1?" with a base case of zero people in front of me (the front row).

2. Considering the process of each person asking the person in front of them the same question, we can see that a recursive approach will work its way down to a base case (row 1 in this example) and then "unwind" back through all the recursive requests to return to the original requester who now has the answer.

3. At each step, the process is the same, each requester has to add one to the answer from the person in front of them.

One clarification related to the movie theater row example. The description involved multiple people, each running the same "code" which asked the person in front for their row number and added 1 to the answer before responding to the person behind them. However, when we use a recursive process in our program, there is only the one computer and program keeping track of all the repeating steps.

6.2 Fibonacci sequence

Let's apply this concept in a mathematical context. One mathematical problem that is easily solved using a recursive approach is calculating Fibonacci numbers. As a refresher, the Fibonacci sequence starts out with the values 0 and 1. It then continues by adding the last two numbers of the current sequence to arrive at the next number in the sequence.

Specifically, starting with 0 and 1 as the sequence, we add them resulting in 1, which becomes the new end of the sequence. The last two numbers are now 1 and 1 which when added together are 2, which again becomes the last number in the sequence. The last two numbers are now 1 and 2 which sum to 3. This continues, 2 plus 3 is 5, 3 plus 5 is 8...

Here are the first 12 numbers in the Fibonacci sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

Thinking about how this is calculated, we can break the problem down into a process of adding together the last two numbers in the current sequence, which in turn adds the previous last two numbers in the sequence, and continuing on until we have as much of the sequence as desired.

To create a function that is recursive we have to be sure to answer three questions, matching the key aspects we discussed in the movie theater row calculating example:

1. What is the base case and how is the function written to identify it?

2. What information does the recursive function need in order to calculate the new value?

3. When is the process complete so that the function knows to stop recursing?

In the case of the Fibonacci sequence we would answer those questions as follows:

1. The base case is that the first value in the sequence is 0 and the second value in the sequence is 1.

2. The function will need to add the last two numbers currently in the sequence if the base case does not apply.

3. The function will need to be told which element of the sequence to compute.

Here is how we could declare our `fibonacci` function, which accounts for the argument value we need (element number in the sequence we are to calculate):

```
def fibonacci(position):
```

Next we need to write the code to apply the base case. In other words, we know that the Fibonacci value for `position` 0 is the value 0 and the Fibonacci value for `position` 1 is the value 1:

```
def fibonacci(position):
    if position == 0:
        return 0
    if position == 1:
        return 1
```

Finally we need to add the **recursive** part of the process. In this case we know that to compute the third value in the sequence we add the second and first values. For the fourth value we add the third and second values. More generally, for the `nth` value we add the `nth - 1` and `nth - 2` values. In code we would write the following to represent that logic:

```
return fibonacci(position - 1) + fibonacci(position - 2)
```

What this says is that to calculate the Fibonacci value at a `position` we will return the result of calling the `fibonacci` function passing `position - 1` and adding to that the value of calling the `fibonacci` function passing the value `position - 2`.

Here is a complete version of a program implementing our recursive Fibonacci algorithm in a function called `fibonacci`. The code calls the `fibonacci` function passing the values 0 through 9 to display the first 10 values of the Fibonacci sequence[4]:

```
def fibonacci(position):
    if position == 0:
        return 0
    if position == 1:
        return 1

    return fibonacci(position - 1) + fibonacci(position - 2)

pos = 0
while pos < 10:
    print(fibonacci(pos), end = ',')
    pos = pos + 1
```

The program produces the following output:

```
$ python fibonacci.py
0,1,1,2,3,5,8,13,21,34,
```

It may take some time studying and thinking through what is happening when the `fibonacci` function runs and calls itself to really grasp how this approach works.

One way to picture the code running is to imagine a stack of plates such as you might see at a salad bar. As new plates are added, the ones already there are pushed down. We call this a **Last-in, First-out** stack. The last plate added will be the first one removed. When we write recursive functions we are creating such a stack of values in the computer.

When a function calls itself, the current set of parameters and local variables are ”pushed down” and temporarily replaced by a new set of parameters and local variables which the recursive call uses. Once the recursive call ends, those original values are ”popped up” and put back for the original function call to continue using.

Fundamentally, each call to a function creates its own unique set of parameters and local variables which are separate from the parameters and local variables used by any other calls to the same function. This happens for each call, so if we call a function 5 times there will be five independent sets of parameters and local variables created for that function.

Let’s walk through the calculation of one value in the Fibonacci sequence using our `fibonacci` function and the concept of a stack. In this example we will look at what happens when we ask for the fifth value in the sequence, which, as we saw above, is 3.

[4]Remember sequence position 0 contains the first value.

We call the `fibonacci` function passing the number 4 as the `position` argument.[5] Inside the function this means that the parameter `position` has the value 4. Looking at the logic in the `fibonacci` function, the two Boolean tests will be false (4 isn't 0 and it isn't 1) and we'll end up at the recursive statement:

```
return fibonacci(position - 1) + fibonacci(position - 2);
```

This means that we will call the `fibonacci` function passing the value 3 (e.g. 4-1). This is a recursive request, the function is calling itself. The currently executing copy of the `fibonacci` function (with `position` = 4) will wait until it gets the answer back from its recursive calls.

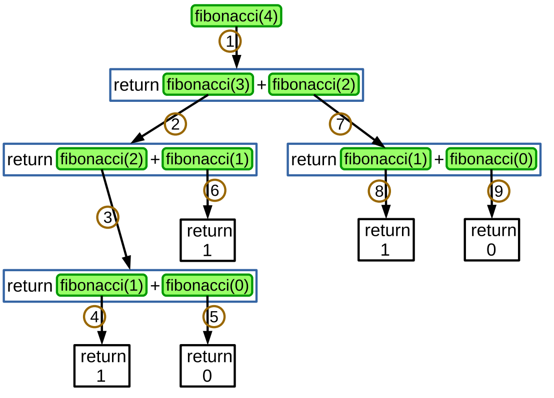
We now start executing a new copy of `fibonacci` passing 3 as the value for `position`. Again, we skip past the two Boolean tests and end up at the recursive call. We then recurse again, passing 2 as the value for `position`. This new executing copy of the `fibonacci` function will wait until it gets the answer back from its recursive call. We now have two copies of the `fibonacci` function waiting on results from their recursive calls.

Eventually we'll end up calling `fibonacci` with the value 1, which is a base case. The function returns the number 1 back to its caller. This allows the previous executing copy of `fibonacci` to continue running, using the returned value. This process continues to unwind its way back to the top of the stack, which is the original call made to `fibonacci` when the program started running.

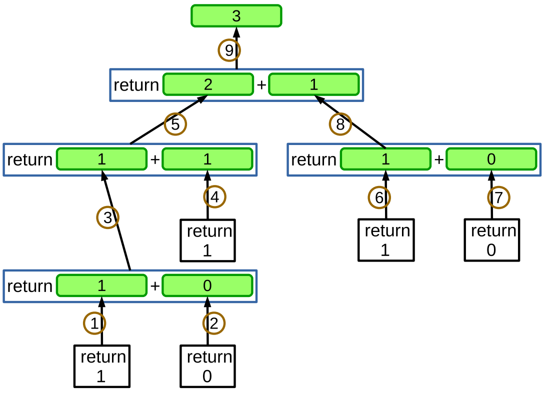
Thinking back to the movie row example, this is similar to how each person asking the person in front of them for their row number has to wait for that person to get an answer from the person in front of them before they will respond to the person behind them. Eventually the request reaches a person in the front row, the base case, who immediately responds "1".

[5] We want the fifth value in the sequence. Remember the first value in the sequence is in the zero-eth position.

Here are two depictions of the entire recursive process for our Fibonacci example where we ask for the fifth element in the sequence. The first depiction illustrates the set of recursive calls showing the `position` value being passed into the function with each call. The numbers in the circles depict the order in which these recursive calls occur when the program runs.



This next image depicts the value being returned by each call and the order in which those returns occur. You'll note that the initial call (where we pass in the argument 4 asking for number in the fifth position of the sequence) contains the number 3, which is the fifth number in the sequence. That is calculated based on the summing of the values from the recursive calls right below, which use the sums of values from the recursive calls below them.



6.3 Searching an ordered list

Another example of a common recursive process is searching an ordered list for a specific value. For example, imagine we are creating a program to check the spelling of words. We already have an list of words that is stored in alphabetical order. Our program needs to accept a word from the user and tell him or her whether it is spelled correctly, or at least if it is a word that is in our list of words.

One way we could write the search would be to start at the beginning of the list and hunt through until we either find a match or arrive at the end of the list. If we find a match we can stop searching and tell the user that the word is spelled correctly. If we reach the end of the list we know the word isn't in our list and we tell the user we can't find that word.

This is a fairly inefficient approach to searching the list. If there are 1,000,000 (one million) words in the list, we will have to hunt through all million words in order to determine that a word is misspelled. If we wanted to use this to spell check a term paper, it would take a long time to run. Is there a better way?

Consider how you would do this task manually, perhaps using a dictionary. You would not start reading words at the beginning and go sequentially through until you matched the word or got to the end of the dictionary. Instead, what would you do? You would jump to the section of the dictionary that begins with the same letter that the word begins with. You would then jump several pages forward, check if the word was further on or earlier and jump forward or back in smaller and smaller increments as appropriate.

With this approach when are you done? Just like the earlier sequential approach, if you find a match you are done and you know the word is spelled correctly. What about the case where the word isn't in the dictionary? You know you are done when you find the location where the word would be if it were there. In other words, when you find two words next to each other where the first comes before the word you seek and the second comes after, you know that your word isn't present.

6.4 Binary search

To search this way using our ordered list example with a computer program we have to make one simplification to the process just described. As humans looking at a dictionary, which is generally organized to make jumping to a specific first letter easy, we can get to the correct part of the dictionary very fast. Our list, however, doesn't have a way to know where each set of words beginning with a specific letter starts. Instead we'll have the computer jump to the middle of the list and check for which of three conditions is true:

1. The word in the middle of the list matches the word we are searching for
2. The word in the middle of the list comes before (in alphabetical order) the word we are searching for
3. The word in the middle of the list comes after the word we are searching for

If the word is a match, we are done. We found the word and can return a result indicating the word is spelled correctly.

What about the other two cases? Based on our comparison with the word in the middle of the list we know whether the word we are searching for comes before or after the word at that location in the list.

We then logically divide the list into two parts, the set of words in the first half of the list and the set of words in the second half of the list. What we are creating are two **sublists**, each containing half of the words that were in the original list.

If the word we seek comes after the word we found in the middle of the original list then we know that our word, if it is present in the list at all, would have to be in the second sublist (remember the list is alphabetically ordered). In this case we do not need to look at the first sublist at all, our word can't be there.

We then treat that sublist as our list to be searched and repeat the process. We jump to the middle of that sublist and check to see which of the three conditions (listed above) are met.

In terms of set size, after our first comparison we have divided the number of words to be searched in half. After the second comparison we have reduced it to one-quarter. The third comparison reduces the amount of the list remaining to one-eighth of the original. You may notice a pattern here. With each comparison we are reducing the remaining amount of the list to search by a factor of 2 of what it was previously.

It turns out this approach of searching is amazingly efficient. In fact, to search our 1,000,000 words, the worst case number of comparisons is the square root of 1,000,000. That turns out to be 1,000. Remember though, the key to being able to do this is to have the list elements in ascending order. The name of this form of search is known as a **binary search**.

We use the term *binary* for the search because it continually divides the remainder of the set in half, creating two sublists. One sublist may hold a matching value and the other can be ignored.

How does this relate to recursion? Reviewing the process of a *binary search* you'll note that it is carrying out the same operation on smaller subsets of the data. The operation always jumps to the middle of the remaining sublist, checks whether the value matches the one we seek or, if it doesn't match, figures out if the value would be earlier or later in the sublist.

The stop conditions are reached when either:

1. A match is found
2. The subdividing of the list has reduced down to a subset with one element and that element's value does not match the word we are seeking

For this example, the information a recursive function would need includes the sorted list of words, the current start and end positions of the sublist being searched and the word being sought.

The function would check the center of the sublist for the word. If the function finds a match it is done and returns a success result, either a Boolean `true` or perhaps the list index location containing the word.

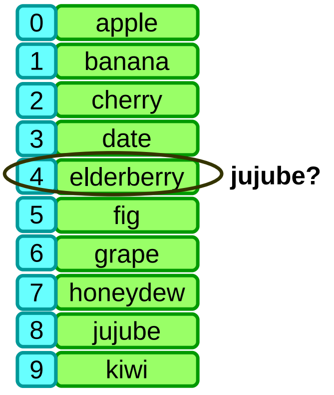
If the word is not matched at the center of the sublist, the function will check whether the word is alphabetically greater or less than the word found in the middle of the list. It then makes a recursive call, replacing the beginning or ending value of the sublist with the location of the old center, offset by one. For example, if the word being sought is greater than the one found then the recursive call needs to check the elements from the old center plus one to the end of the sublist.

6.5 Depicting a successful binary search

Let's illustrate the process of a binary search without worrying about programming it yet. First, we'll work through a depiction of the process where the value we are searching for is in the list.

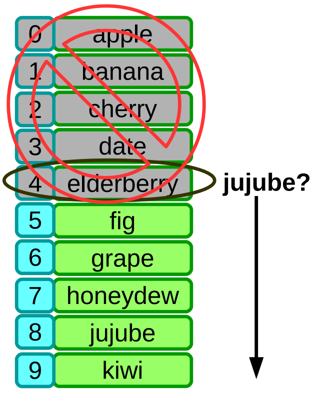
For our example, we start with an list containing 10 fruit names in alphabetical order. In the image we see each word and to the left the element number in the list which contains that value. The list has 10 values so the list elements are numbered 0 to 9.

We want to see if the word **jujube** matches one of our words. We start by finding the middle of the list. To find the middle we add the starting index value to the ending index value and divide by 2. For this process we always use integer division (no fractional part). Given our 10 element list we calculate `(0 + 9) / 2` resulting in the integer value `4`. We compare our search term, **jujube**, to the word located in element 4, which is **elderberry**.

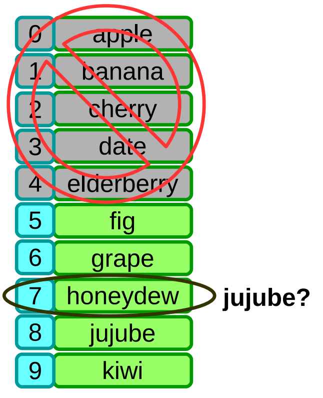


The words do not match, so we check to see if **jujube** comes before or after **elderberry**. It comes after, meaning that if it is in the list it must be in an element further down the list. All the elements before the one containing **elderberry**, as well as the element containing **elderberry**, cannot contain our word and do not need to be looked at.

Since we know the word would come after the position just checked, we change the starting position to be one greater than the position just checked, which is the value 5.

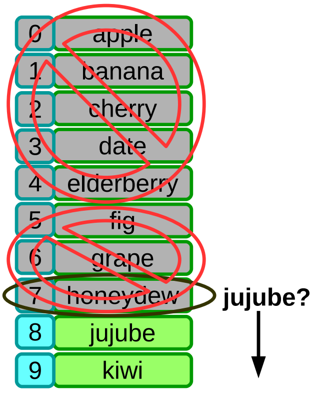


Now we need to repeat this process on the remaining sublist. We calculate the middle element of the remaining elements. Our updated starting element number (as described above) is 5 (one past elderberry) and the end is still 9. Our calculation is then `(5 + 9) / 2` which results in `7`. We compare our search term, **jujube**, to the word located in element 7, which is **honeydew**.

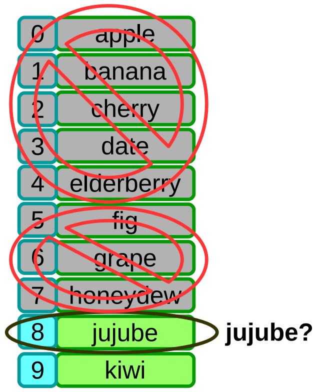


The words do not match, so we check to see if **jujube** comes before or after **honeydew**. It comes after, meaning that if it is in the list it must be in an element further down the list. All the elements before the one containing **honeydew**, as well as the element containing **honeydew** cannot contain our word and do not need to be looked at.

Since we know the word would come after the position just checked, we again change the starting position to be one greater than the position just checked, which is the value 8.



Again we repeat this process on the remaining sublist. We calculate the middle element of the remaining elements. Our new starting element number is 8 and the end is still 9. Our calculation is then `(8 + 9) / 2` which results in `8`. We compare our search term, **jujube**, to the word located in element 8, which is **jujube**.

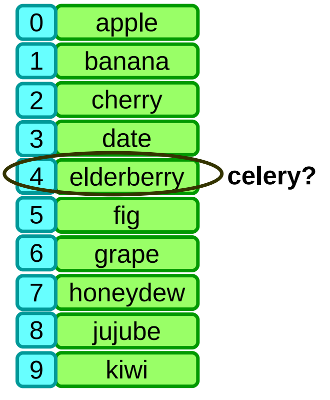


The words match! The search is completed having made only 3 comparisons in our list of 10 elements.

6.6 Depicting an unsuccessful binary search

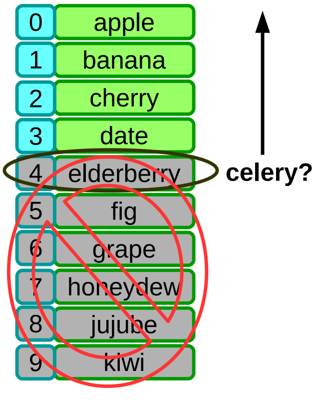
Here is a depiction of the binary search process where the value we are searching for is *not* in the list. We start with the same list of 10 fruit names in alphabetical order.

We want to see if the word **celery** matches one of our words. As before, we start by finding the middle of the list, element 4. We compare our search term, **celery**, to the word located in element 4, which is **elderberry**.

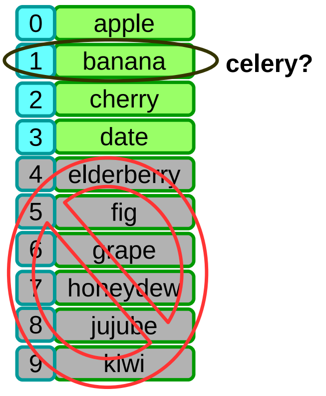


The words do not match, so we check to see if **celery** comes before or after **elderberry**. It comes before, meaning that if it is in the list it must be in an element earlier in the list. All the elements after the one containing **elderberry**, as well as the element containing **elderberry**, cannot contain our word and do not need to be looked at.

Since we know the word would come before the position just checked, we change the ending position to be one less than the position just checked, which is the value 3.

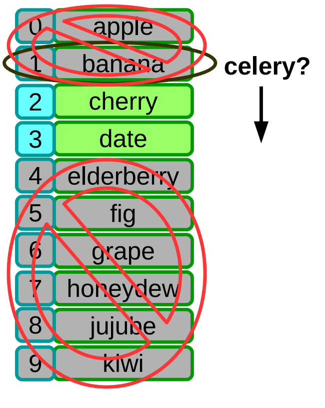


Again, we need to repeat this process on the remaining sublist. We calculate the middle element of the remaining elements. Our updated ending element number (as descrbied above) is 3 (one before elderberry) and the beginning is still 0. Our calculation is then `(0 + 3) / 2` which results in `1`. We compare our search term, **celery**, to the word located in element 1, which is **banana**.

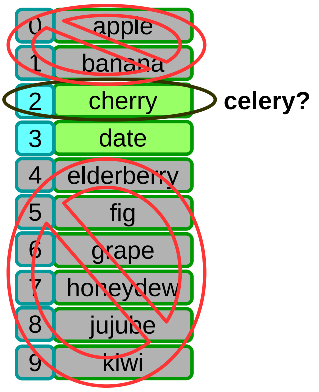


The words do not match, so we check to see if **celery** comes before or after **banana**. It comes after, meaning that if it is in the list it must be in an element further down the list. All the elements before the one containing **banana**, as well as the element containing **banana** cannot contain our word and do not need to be looked at.

Since we know the word would come after the position just checked, we change the starting position to be one greater than the position just checked, which is the value 2.

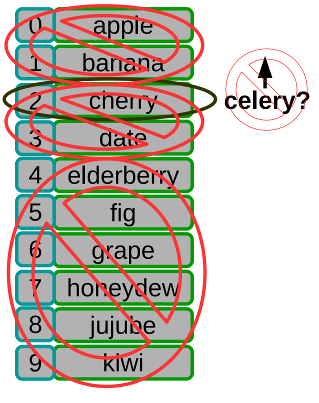


As usual, we repeat the process on the remaining sublist. We calculate the middle element of the remaining elements. Our updated starting element number is 2 and the end is still 3. Our calculation is then `(2 + 3) / 2` which results in `2`. We compare our search term, **celery**, to the word located in element 2, which is **cherry**.



The words do not match, so we check to see if **celery** comes before or after **cherry**. It comes before, meaning that if it is in the list it must be in an element earlier in the list.

Since we know the word would come before the position just checked, we change the ending position to be one less than the position just checked, which is the value 1.



We now find that the starting element position is 2 and the ending element position 1. Since start is greater than end we know the word can't be in the list.

Fundamentally, ***all the earlier elements and all the later elements have been ruled out***. Therefore the word must not be in the list. We have completed the search having made only 3 comparisons in our list of 10 elements.

6.7 Implementing a recursive binary search

Let's look at the code to implement the binary search process we just described. We'll create a recursive function called `binary_search` to implement the logic.

There are four pieces of information the function needs:

1. The ordered list of words

2. The word we are searching for
3. The current starting list element number
4. The current ending list element number

In the binary search process we will first calculate the center element. We will also need a variable to house the result of comparing the word being sought to the word in the center element of the sublist.

Next we need to compare the words (strings). Remember from our discussion of strings that we can compare strings using the relational operators.

The next part of the function will need to evaluate the result of the comparison.

1. If we have found a match we'll return the list index position of the element containing the word.
2. If the search term is less than the value in the list element we will update the ending list element number to be one less than the current center position (e.g. we know the word has to be earlier in the list).
3. If the search term is greater than the value in the list element we will update the starting list element number to be one more than the current center position (e.g. we know the word has to be later in the list).

Finally we need to determine whether we have reached the end of the process, meaning there is no smaller subset of the original list left to search. The easiest way to figure that out is to compare the updated start and end positions. If they cross, meaning that the start position becomes greater than the end position, we have finished the search and not found a match. If the start position is equal to or less than the end position we need to recurse and check the new sublist.

Here is the entire function, which implements the process just described. Make sure that each part makes sense and that you can see how the code relates to the explanation above.

```
# Conduct a binary search
# Two base cases:
#    1. Match found
#    2. No match found
def binary_search(wordlist, searchterm, startpos, endpos):
    checkpos = (startpos + endpos) // 2
    if searchterm == wordlist[checkpos]:
        # Words match - return the matching position
        return checkpos
    elif searchterm < wordlist[checkpos]:
        # search term comes earlier in list
        # change ending position
        endpos = checkpos - 1
    else:
```

```
        # search term comes later in the list
        # change starting position
        startpos = checkpos + 1

    # See if there are still more words to check
    if startpos <= endpos:
        # Still more words to check, make recursive call
        return binary_search(wordlist, searchterm, startpos, endpos)
    else:
        # No more words to check, the word is not in the list
        return -1

fruits = [ 'apple', 'banana', 'cherry', 'date', 'elderberry',
    'fig', 'grape', 'honeydew', 'jujube', 'kiwi' ]
position = binary_search(fruits, 'jujube', 0, len(fruits) - 1)
print('Found jujube?', position)

print('===============================')

position = binary_search(fruits, 'carrot', 0, len(fruits) - 1)
print('Found carrot?', position)
```

When we run this we see the following output:

```
Found jujube? 8
===============================
Found carrot? -1
```

This means that *jujube* was found in element 8 of the `fruits` list and *carrot* was not found in the list (we chose to use -1 as the signal for not finding a match).

Let's add a few print statements so that we can see what is happening when the `binary_search` function runs. Here is an updated version that includes some `print` output to display the variable values as the recursive calls are made.

```
# Conduct a binary search
# Two base cases:
#    1. Match found
#    2. No match found
def binary_search(wordlist, searchterm, startpos, endpos):
    checkpos = (startpos + endpos) // 2

    print('binary_search called for searchterm:' + searchterm +
        ' startpos:' + str(startpos) + ' endpos:' + str(endpos))
    print('checkpos:' + str(checkpos) + ' word at list element:' +
        wordlist[checkpos] + '\n')

    if searchterm == wordlist[checkpos]:
        # Words match - return the matching position
        return checkpos
    elif searchterm < wordlist[checkpos]:
        # search term comes earlier in list
        # change ending position
        endpos = checkpos - 1
    else:
```

```
        # search term comes later in the list
        # change starting position
        startpos = checkpos + 1

    # See if there are still more words to check
    if startpos <= endpos:
        # Still more words to check, make recursive call
        return binary_search(wordlist, searchterm, startpos, endpos)
    else:
        # No more words to check, the word is not in the list
        return -1

fruits = [ 'apple', 'banana', 'cherry', 'date', 'elderberry',
    'fig', 'grape', 'honeydew', 'jujube', 'kiwi' ]
position = binary_search(fruits, 'jujube', 0, len(fruits) - 1)
print('Found jujube?', position)

print('===============================')

position = binary_search(fruits, 'carrot', 0, len(fruits) - 1)
print('Found carrot?', position)
```

When we run this we see some details about how the recursive calls are operating:

```
$ python binary_search_addloutput.py
binary_search called for searchterm:jujube startpos:0 endpos:9
checkpos:4 word at list element:elderberry

binary_search called for searchterm:jujube startpos:5 endpos:9
checkpos:7 word at list element:honeydew

binary_search called for searchterm:jujube startpos:8 endpos:9
checkpos:8 word at list element:jujube

Found jujube? 8
===============================
binary_search called for searchterm:carrot startpos:0 endpos:9
checkpos:4 word at list element:elderberry

binary_search called for searchterm:carrot startpos:0 endpos:3
checkpos:1 word at list element:banana

binary_search called for searchterm:carrot startpos:2 endpos:3
checkpos:2 word at list element:cherry

Found carrot? -1
```

Make sure to study the output and trace through the recursive function calls that are executed in these examples. Drawing a table of values and keeping track of the *stack* of parameter and local variable values may be helpful as well.

6.8 Glossary

recurse: Having a function call itself, typically with a subset of the data that was provided to the initial function call.

recursion: The use of a function that uses its own definition to apply the same logic over and over. Typically this is used to break a large problem into smaller pieces and ultimately solve the larger problem by combining the results of the recursive calls.

recursive function: A function that uses recursion.

stack: A data structure where information (data) is placed in an ordered list. In the case of recursion, the stack holds the data from earlier calls to a recursive function that are waiting on results from their recursive calls. As recursive calls end, the previous data is retrieved from the stack and restored for the earlier function call to use.

6.9 Exercises

Exercise 6.1 Write a program that takes an ordered list of integers and uses a binary search to find a specific number. Assume the numbers are all positive allowing you to use -1 as the signal for not finding a match. You should be able to use the example WordSearch class from the chapter as a starting point for this program.

Exercise 6.2 Write a program that prompts the user for a positive number and then sums all the numbers from 1 up to the entered value. The program **may *not* use a loop**. Rather, it must use a recursive function to carry out the summing process. Here is an example of what a run of the program would look like:

```
Enter a positive number: 5
Result: 15
```

Exercise 6.3 Sometimes when programmers get bored or want to have a bit of fun, they add a harmless **Easter Egg**[6] to their program. Modify the program from the previous exercise so that it prints a funny message when the user types a specific value. For example you might have the program print the message `Answer to the Ultimate Question of Life, the Universe, and Everything` when the user enters the number 42. The program should behave normally for all other supplied values. Here are a couple of sample executions of the program:

```
Enter a positive number: 42
Answer to the Ultimate Question of Life, the Universe, and Everything
Result: 903

Enter a positive number: 7
Result: 28
```

We are not encouraging you to put Easter Eggs in your programs—this is just an exercise.

[6] en.wikipedia.org/wiki/Easter_egg_(media)

Chapter 7

Strings

7.1 A string is a sequence

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement copies the character at index position 1 from the `fruit` variable and assigns it to the `letter` variable.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

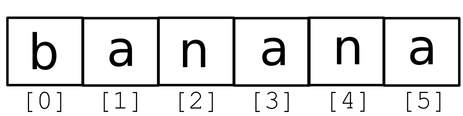
But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of `'banana'` is `b`, not `a`. But in Python, the index is an offset from the beginning of the string, and the **offset of the first letter is zero**.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So `b` is the 0th letter ("zero-eth") of `'banana'`, `a` is the 1th letter ("one-eth"), and `n` is the 2th ("two-eth") letter.



You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get an error:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

7.2 Getting the length of a string using `len`

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

7.3 Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**.

One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Another way to write a traversal is with a `for` loop:

```
for char in fruit:
    print(char)
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

7.4 String slices

A segment of a string is called a **slice** or **substring**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first **but excluding the last**.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

When you take a slice from a string, the slice is a new string. It contains a copy of the characters from the range of the slice.

The slice (or substring) convention of including the first index character and excluding the last is common when working with strings in many languages. There are two advantages to this approach. First, you may use the length of the string as the second value in the slice to get the last character in the string. Second, if you subtract the starting index from the ending index the result is the number of characters placed in the slice.

7.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

7.6 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result—the total number of `a`'s.

7.7 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

7.8 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

To understand how we could convert a string to lowercase characters we need to delve into accessing a string's methods.

7.9 `string` methods

Strings are an example of Python **objects**. An object is also known as an **instance**. Each object contains data (in this case the actual string itself) and has access to **methods**, which are effectively functions that are built for each instance to use. The data and methods are defined by the type[1].

We can list the methods available to an object (instance) using Python's `dir` function, which lists the methods available for an object. We've already seen that the `type` function shows the type of an object. We can use the `help` function to read documentation for a method, if the programmer of the method included such documentation.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
```

[1] We will spend a great deal of time later in the semester creating our own types and objects

```
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

While the `dir` function lists the methods, and you can use `help` to get some simple documentation on a method, a better source of documentation for string methods would be https://docs.python.org/3/library/stdtypes.html#string-methods.

Calling a **method** is similar to calling a function—it takes arguments and returns a value—but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method `upper` returns a new string with all uppercase letters using the string it is run on as the original value:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word` object (instance).

For example, there is a string method named `find` that searches for the position of one string within another:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

The `find` method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the `strip` method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as **startswith** return boolean values.

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

You will note that `startswith` requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the `lower` method.

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

In the last example, the method `lower` is called and then we use `startswith` to see if the resulting lowercase string starts with the letter "p". As long as we are careful with the order, we can make multiple method calls in a single expression.

7.10 Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

```
From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008
```

and we wanted to pull out only the second half of the address (i.e., `uct.ac.za`) from each line, we can do this by using the `find` method and string slicing.

First, we will find the position of the at-sign (@) in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

We use a version of the `find` method which allows us to specify a position in the string where we want `find` to start looking. When we slice, we extract the characters from "one beyond the at-sign through up to *but not including* the space character".

The documentation for the `find` method is available at `https://docs.python.org/3/library/stdtypes.html#string-methods`.

7.11 Format strings

A **format string** contains text and **replacement fields**, which are delineated with curly braces, {}. Format strings allow us to create strings that combine text with other values, including literals and variables.

As an example, if we were writing a game and wanted to display the current score we might display the message:

```
Current score is 5
```

However, the score value, 5 as shown above, can't be part of the hardcoded string's text. Instead it is probably being stored in a variable, since it can change as the user plays the game. The current score needs to be included in the string where the 5 is shown.

That's where a format string becomes useful, allowing us to combine text with replacement fields. Here is a format string that could be used for the score example. It starts with the text `Current score is` and then includes a replacement field to specify that some value should be inserted at that point in the string:

```
'Current score is {}'
```

Later, we'll see that we can include values in the replacement field to control the formatting of data. If the braces are left empty, {}, the default representation for the data will be used. In all cases, the result is a string.

We call the `format` method to provide the values used in place of the replacement fields in the format string. Let's put the score example together:

```
>>> score = 6
>>> message = 'Current score is {}'.format(score)
>>> print(message)
Current score is 6
```

The `message` variable is assigned a value using a format string. The resulting value is `'Current score is 6'`, where the `6` is the `score` variable's value being passed to the `format` method. If you think about all the programs that you use and all the ways they show you information, mixing data with text, this technique becomes very handy.

We can place more than one replacement field in a string. Here is a basic example which uses two variables' values as part of the string being printed:

```
>>> camels = 42
>>> elk = 25
>>> message = 'There are {} camels and {} elk.'.format(camels, elk)
>>> print(message)
There are 42 camels and 25 elk.
```

Again, the `message` variable is assigned a value using a format string. The resulting value is `'There are 42 camels and 25 elk.'`, where `42` and `25` are values taken from the variables passed to the `format` method.

Replacement fields may appear anywhere in a format string, so you can embed values anywhere in a sentence. Here is an example using three replacement fields, one at the begining, one near the middle, and one at the very end:

```
>>> label_id = 1
>>> camels = 42
>>> mood = 'happy'
>>> print('{}: The {} camels are {}'.format(label_id, camels, mood))
1: The 42 camels are happy
```

Values passed to the `format` method may be any sort of data, such as integers, floats, and strings. By default, for each replacement field in the string, there must be a variable passed to `format`. The values are used in left to right order.

You can control the formatting of data as it is included in the string. For instance, you can limit the number of decimal places displayed for a number or set the value to be left or right justified. Formatting is controlled by adding **format specifications** within the curly braces of the replacement field.

Let's demonstrate some formatting capabilities. We'll begin by formatting numeric values, limiting the width of a value and the number of decimal places displayed.

To set the desired width (number of characters occupied by the number) and the number of decimal places, we define a format specification starting with a colon, followed by the total number of characters to display (width), a dot, the number of digits after the decimal point to display, and the letter **f**, indicating a floating point value.

Our next examples will set the number of decimal places without limiting the width (we'll place no number before the dot in the format specification of the replacement field). Look carefully at the output and be sure you understand what is happening.

```
>>> print('{:.0f}'.format(3000.14159))
3000
>>> print('{:.1f}'.format(3000.14159))
3000.1
>>> print('{:.2f}'.format(3000.14159))
3000.14
>>> print('{:.5f}'.format(3000.14159))
3000.14159
>>> print('{:.10f}'.format(3000.14159))
3000.1415900000
>>>
```

Now let's try setting the width as well. *Note that the width is a minimum.* If the whole number portion of a value is too large for the width, it will not be truncated. Instead, the width will be increased to include all the digits to the left of the decimal point.

```
>>> print('{:1.0f}'.format(3000.14159))
3000
>>> print('{:4.0f}'.format(3000.14159))
3000
>>> print('{:10.0f}'.format(3000.14159))
      3000
>>> print('{:10.2f}'.format(3000.14159))
   3000.14
```

Why do the last two examples start with empty spaces? Hint: the third printed value contains 6 spaces before the 3000 and the fourth printed value contains 3 spaces before the 3000.14

You can format string values in a similar way, controlling the width and also the text justification (left, right, and centered). The characters >, <, and ^ are used to control right, left, and center justification, respectively. Justification works with numeric values as well. As with numbers, the width given in the replacement field is the minimum number of characters Python will use. If the string is longer, then the width will grow to accomodate.

```
>>> print('{:2}'.format('Hi!'))
Hi!
>>> print('{:3}'.format('Hi!'))
Hi!
>>> print('{:20}'.format('Hi!'))
Hi!
>>> print('{:<20}'.format('Hi!'))
Hi!
>>> print('{:>20}'.format('Hi!'))
                 Hi!
>>> print('{:^20}'.format('Hi!'))
```

```
        Hi!
```

Based on the last 4 print statements, what is the default justification for text values? Now review the numeric formatting we discussed earlier and determine what the default justification is for numbers.

If you include the precision when formatting a string, then the string **will be truncated** if it is longer than the precision value. For example:

```
>>> print('{:.3}'.format('Hi!'))
Hi!
>>> print('{:.2}'.format('Hi!'))
Hi
>>> print('{:20.2}'.format('Hi!'))
Hi
>>> print('{:>20.2}'.format('Hi!'))
                  Hi
>>> print('{:^20.1}'.format('Hi!'))
         H
```

Again, review the replacement field and the output to be sure you can explain what is happening.

A couple of common errors you may encounter with format strings are having more replacement fields than values being passed to the `format` method, and defining the replacement field for one type of information but passing a different type of value to the `format` method. Here are examples of those situations:

```
>>> print('{} {} {}'.format('Hello', 'World!'))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: Replacement index 2 out of range for positional args tuple
>>> print('{:.2f}'.format('Hi'))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'f' for object of type 'str'
```

In the first example, there aren't enough values being passed, so Python displays an **IndexError**. In the second example, the value being passed is the wrong type; the replacement field was defined for a floating point value and the `format` method is being passed a string, so Python displays a **ValueError**.

Format strings and replacement field options are powerful. They allow you to make the output of your Python programs look organized and professional. However, they can be overwhelming as you start using them. Be patient, they'll become second nature as you write more sophisticated programs. You can read more about format strings at https://docs.python.org/3/library/string.html#formatstrings

7.12 Debugging

A skill that you should cultivate as you program is always asking yourself, “What could go wrong here?” or alternatively, “What crazy thing might our user do to crash our (seemingly) perfect program?”

For example, look at the program which we used to demonstrate the `while` loop in the chapter on iteration:

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print(line)

print 'Done!'
```

Look what happens when the user enters an empty line of input:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#' :
```

The code works fine until it is presented an empty line. Then there is no zero-th character, so we get a traceback. There are two solutions to this to make line three “safe” even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

```
    if line.startswith('#') :
```

Another way is to safely write the `if` statement using the **guardian** pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
    if len(line) > 0 and line[0] == '#' :
```

7.13 Glossary

counter: A variable used to count something, usually initialized to zero and then incremented.

empty string: A string with no characters and length 0, represented by two quotation marks.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

format string: A string, used with the format operator, that contains format sequences.

flag: A boolean variable used to indicate whether a condition is true.

invocation: A statement that calls a method.

immutable: The property of a sequence whose items cannot be assigned.

index: An integer value used to select an item in a sequence, such as a character in a string.

item: One of the values in a sequence.

method: A function that is associated with an object and called using dot notation.

object: Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

search: A pattern of traversal that stops when it finds what it is looking for.

sequence: An ordered set; that is, a set of values where each value is identified by an integer index.

slice: A part of a string specified by a range of indices.

traverse: To iterate through the items in a sequence, performing a similar operation on each.

7.14 Exercises

Exercise 7.1 Given that `fruit` is a string, what does `fruit[:]` mean?

Exercise 7.2 Write a `while` loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line, except backwards. A similar program can be found on page 92.

Exercise 7.3 Encapsulate the code from the letter counting example program (page 94) in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.

You should then call the function a couple of times passing a different string and letter to count. Make sure the results are correct.

Exercise 7.4 There is a string method called `count` that is similar to the function shown in the example on page 97. Read the documentation of this method at https://docs.python.org/3/library/stdtypes.html#string-methods and write an invocation that counts the number of times the letter `a` occurs in `'banana'`. Be sure your code works regardless of the case of the letters. That means the count should work for `A` and `'banANA'` as well as `a` and `'BANAna'`. In all cases it should find 3.

Exercise 7.5 Take the following Python code that stores a string:

```
spam_score = 'X-DSPAM-Confidence: 0.8475'
```

Use `find` and string slicing to extract the portion of the string after the colon character and then use the `float` function to convert the extracted string into a floating point number. Print the resulting score to three decimal places using the format() method.

Exercise 7.6 Read the documentation of the string methods at https://docs.python.org/3/library/stdtypes.html#string-methods. You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

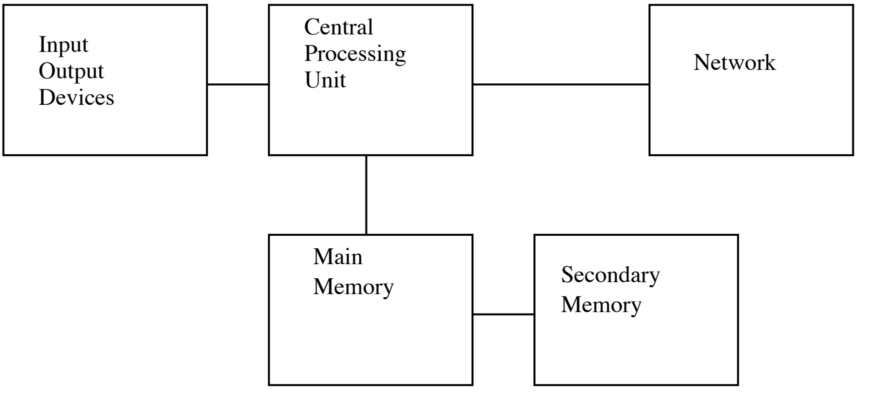
Chapter 8

Files

8.1 Persistence

So far, we have learned how to write programs and communicate our intentions to the **Central Processing Unit** using conditional execution, functions, and iterations. We have learned how to create and use data structures in **Main Memory**. The CPU and main memory are where our software works and runs. It is where all of the "thinking" happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient *and fun* exercises to learn Python.



In this chapter, we start to work with **Secondary Memory** (or files). Secondary memory is not erased when the power is turned off. In the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

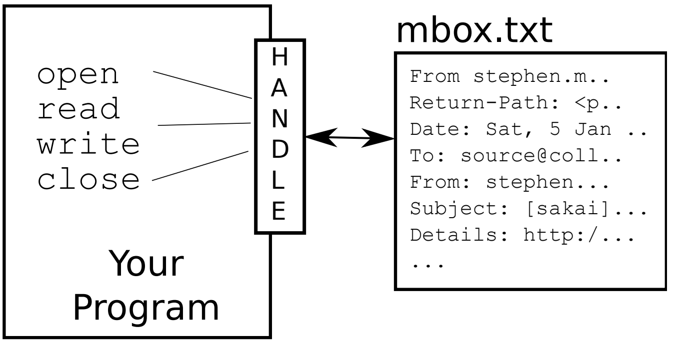
We will primarily focus on reading and writing text files such as those we create in a text editor. Later we will see how to work with binary files specifically designed to be read and written through specialized software.

8.2 Opening files

When we want to read or write a file (perhaps on your hard drive), we first must **open** the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file `mbox.txt`[1], which should be stored in the same folder that you are in when you start Python.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

If the `open` is successful, the operating system returns a **file handle**. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.



If the file does not exist, `open` will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'stuff.txt'
```

Later we will use `try` and `except` to deal more gracefully with the situation where we attempt to open a file that does not exist.

8.3 Closing files

Once we are finished using the file we need to **close** it. This means that we tell the computer that we are done reading the file and it may release the file handle that was created to allow us access to it.

[1]You may download the `mbox.txt` file from https://www.monead.com/introcompsci/mbox.txt

This may not seem like a big deal but computers have a limited number of file handles that may be open at one time. If we fail to close files then our program may eventually be unable to open other files once we exhaust the available file handles. Failing to close files falls into a category of programming bugs known as **leaking resources**.

The term **leaking resources** refers to the fact that we are wasting a finite resource, in this case file handles. Other resources that a program may leak include memory and database connections.

To close a file we call the `close()` method on the file handle we have open.

```
>>> fhand.close()
```

Forgetting to close a file makes for a sloppy program. Python does close all open files when the program ends, which often hides our mistake. If your program is short and only opens a few files you usually won't experience a failure. Even so, closing any file when you are done with it is a good habit to establish.

8.4 Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

The entire file of mail interactions is available from `https://www.monead.com/introcompsci/mbox.txt` and a shortened version of the file is available from `https://www.monead.com/introcompsci/mbox-short.txt`. These files are in a standard format for a file containing multiple mail messages. The lines which start with "From " separate the messages and the lines which start with "From:" are part of the messages. For more information about the mbox format, see `en.wikipedia.org/wiki/Mbox`.

To break the file into lines, there is a special character that represents the "end of the line" called the **newline** character.

In Python, we represent the **newline** character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the `\n` in the string, but when we use `print` to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

You can also see that the length of the string `'X\nY'` is *three* characters because the newline character is a single character.

So when we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

8.5 Reading files

While the **file handle** does not contain the data for the file, it is quite easy to construct a `for` loop to read through and count each of the lines in a file:

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
fhand.close()
print('Line Count:', count)
```

When this program runs, we get the following output:

```
Line Count: 132045
```

We can use the file handle as the sequence in our `for` loop. Our `for` loop simply counts the number of lines in the file and prints them out. The rough translation of the `for` loop into English is, "for each line in the file represented by the file handle, add one to the `count` variable."

The reason that the `open` function does not read the entire file is that the file might be quite large with many gigabytes of data. The `open` statement takes the same amount of time regardless of the size of the file. The `for` loop actually causes the data to be read from the file.

When the file is read using a `for` loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the `line` variable for each iteration of the `for` loop.

Because the `for` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `read` method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> fhand.close()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file `mbox-short.txt` are read directly into the variable `inp`. We use string slicing to print out the first 20 characters of the string data stored in `inp`.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable **inp**. Remember that this form of the `open` function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a `for` or `while` loop.

8.6 Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix “From:”, we could use the string method **startswith** to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print(line)
fhand.close()
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? This is due to that invisible **newline** character. Each of the lines ends with a newline, so the `print` statement prints the string in the variable **line** which includes a newline and then `print` adds *another* newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the **rstrip** method which strips whitespace from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print(line)
fhand.close()
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using `continue`. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    # Process our 'interesting' line
```

```
    print(line)
fhand.close()
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with "From:", which we skip using `continue`. For the "interesting" lines (i.e., those that start with "From:") we perform the processing on those lines.

We can use the `find` string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since `find` looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string "@uct.ac.za" (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1 :
        continue
    print(line)
fhand.close()
```

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

8.7 Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using `raw_input` as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
```

```
fhand.close()
print('There were', count, 'subject lines in', fname)
```

We read the file name from the user and place it in a variable named `fname` and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, "What could go possibly wrong here?" or "What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?"

8.8 Using `try`, `except`, and `open`

I told you not to peek. This is your last chance.

What if our user types something that is not a file name?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

Do not laugh, users will eventually do every possible thing they can do to break your programs—either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called **Quality Assurance** (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the `try`/`except` structure. We need to assume that the `open` call might fail and add recovery code when the `open` fails as follows:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
fhand.close()
print('There were', count, 'subject lines in', fname)
```

The `exit` function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Protecting the `open` call is a good example of the proper use of `try` and `except` in a Python program. We use the term "idiomatic" when we are doing something the commonly accepted way in a language. We might say that the above example is the idiomatic way to open and close a file in Python.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is "more idiomatic". The goal to be "more idiomatic" captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

8.9 Writing files

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method of the file handle object puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The `print` statement automatically appends a newline, but the `write` method does not add the newline automatically.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

Although we saw that we can get away with forgetting to close a file we are reading, writing files is a different story. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

8.10 Using `finally` with `close`

Your code for writing files , and sometime reading files, can fail even after the file is open. Consider what would happen if your program was writing to a file and the disk became full. No more writes world work. Python would detect the error and we could handle the situation in the `except` code.

This leads to a conundrum. Our program may work fine, opening and writing the file, or it could open the file but fail when writing, or it could fail when opening the file. In the first two cases we need to close the file. How can we write code that will close the file regardless of whether an error triggered our `except` code?

This is where we can use `finally` in combination with `try` and `except`. The code we include in the `finally` section will run whether or not an error causes our `except` code to run.

You should use the example program, below, as a template for all of your file interactions:

1. Open and process the file using `try`

2. Report errors using `except`. Note in the example how we can get the text of the error.

3. Close the file using `finally`

Note the example reads a file but the same pattern should be used when writing a file.

```
fname = input('Enter the file name: ')
fhand = None
try:
    fhand = open(fname)
    count = 0
    for line in fhand:
        if line.startswith('Subject:') :
            count = count + 1
    print('There were', count, 'subject lines in', fname)
except Exception as err:
    print('File', fname, 'cannot be processed, caused by:', err)
    exit()
finally:
    if fhand != None:
        fhand.close()
```

8.11 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at `wikipedia.org/wiki/Newline`. Or, of course, you could write one yourself.

8.12 Glossary

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

Idiomatic: A technique that works elegantly in a programming language. "Using try, except, and finally is the *Idiomatic* way to recover from missing files in Python".

newline: A special character used in files and strings to indicate the end of a line of text.

Quality Assurance: A person or team focused on insuring the overall quality of a software product. QA is often involved in testing a product and identifying problems before the product is released.

text file: A sequence of characters stored in permanent storage like a hard drive.

8.13 Exercises

Exercise 8.1 Write a program to read through a file and print the contents of the file (line by line) all in upper case. Executing the program will look as follows:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
 BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
 SAT, 05 JAN 2008 09:14:16 -0500
```

You can download the file from https://www.monead.com/introcompsci/mbox-short.txt

Exercise 8.2 Write a program to prompt for a file name, and then read through the file and look for lines of the form:

`X-DSPAM-Confidence:` **0.8475**

When you encounter a line that starts with "X-DSPAM-Confidence:" pull apart the line to extract the floating-point number on the line. Count these lines and then compute the total of the spam confidence values from these lines. When you reach the end of the file, print out the average spam confidence.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Test your file on the `mbox.txt` and `mbox-short.txt` files.

Exercise 8.3 Write a program named long_colon_lines.py to prompt for a file name, and then read through the file and look for lines that are between 40 and 50 characters and contain a colon ":". For each matching line display its line number in the file as well as the text. At the end, report the number of lines that were matched. Executing the program should look as follows:

```
$ python long_colon_lines.py
What file do you want to search? quotes.txt
96: Second childhood: when old goats act like kids
102: Tuning fork: utensil used at musical banquets
104: Checks and balances: famous juggling act
114: Pipedreams: chimney cleaners' nightmares
115: Gerry Mander: an unscrupulous politician
119: Race track: a place where windows clean people
185: Weakdays: after hard nights on the town.
188: Ancient hill top castle: ruin with a view.
189: Beginner's gardening manual: trowel and error.
193: Rhinoceros: Unicorn designed by committee.
199: Bandleader: A man who can face the music.
Number of lines that were matched: 11
```

Test your file on the `quotes.txt` file which you can download from https://www.monead.com/introcompsci/quotes.txt

Chapter 9

Lists

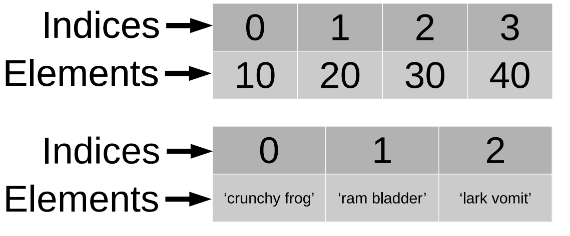
9.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. We can depict these as:



The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

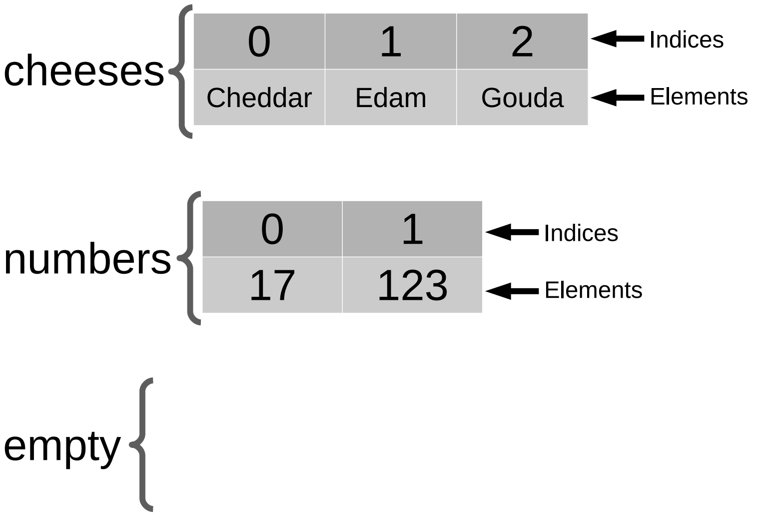
```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```



9.2 Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print(cheeses[0])
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index “maps to” one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.

- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> print(cheeses[2])
Gouda
>>> print(cheeses[-1])
Gouda
>>> print(cheeses[0])
Cheddar
>>> print(cheeses[-3])
Cheddar
```

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

9.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

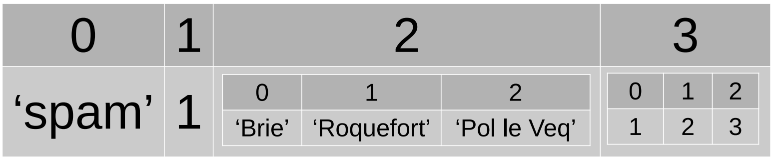
This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n-1$, where n is the length of the list. Each time through the loop, `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never executes the body:

```
for x in empty:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```



You may get the length of a nested list by using the `len()` function with the element containing the nested list You may use additional square brackets to access specific elements of a nested list.

```
>>> nested = ['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
>>> print(len(nested))
4
>>> print(nested[0])
spam
>>> print(nested[2])
['Brie', 'Roquefort', 'Pol le Veq']
>>> print(len(nested[2]))
3
>>> print(nested[2][1])
Roquefort
```

9.4 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the `*` operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1, 2, 3]` three times.

9.5 List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

9.6 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

9.7 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

As usual, the slice selects all the elements up to, **but not including**, the second index.

9.8 Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.666666666666668
```

The `sum()` function only works when the list elements are numbers. The other functions (`max()`, `len()`, etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

```
total = 0
count = 0
while True:
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)
```

In this program, we have `count` and `total` variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

```
numlist = list()
while True:
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)
```

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

9.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter `l` because it looks too much like the number `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Once you have used `split` to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

You can call `split` with an optional argument called a **delimiter** that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. **Note that `join` is a string method**, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

9.10 Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the "interesting lines" and then **parse** the line to find some interesting *part* of the line. What if we wanted to print out the day of the week from those lines that start with "From "?

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

The `split` method is very effective when faced with this kind of problem. We can write a small program that looks for lines starting with "From ", using `startswith`), next we can `split` those lines, and finally print out the third word in the line.

Here are a few lines from the mbox-short.txt[1] file that the program uses:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
From louis@media.berkeley.edu Fri Jan  4 18:10:48 2008
Return-Path: <postmaster@collab.sakaiproject.org>
From zqian@umich.edu Fri Jan  4 16:10:39 2008
Return-Path: <postmaster@collab.sakaiproject.org>
From rjlowe@iupui.edu Fri Jan  4 15:46:24 2008
Return-Path: <postmaster@collab.sakaiproject.org>
```

Here is the program that uses `startswith` and `split` as described above:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])
```

Here we also use the contracted form of the `if` statement where we put the `continue` on the same line as the `if`. This contracted form of the `if` functions the same as if the `continue` were on the next line and indented.

The program produces output that starts like this:

```
Sat
Fri
Fri
Fri
    ...
```

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

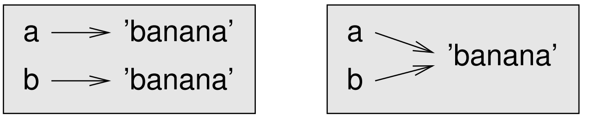
[1]You may download the `mbox-short.txt` https://www.monead.com/introcompsci/mbox-short.txt, which should be stored in the same folder that you are in when you start Python.

9.11 Objects and values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

we know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:



In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the is operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both a and b refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `a = [1,2,3]`, a refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

9.12 Aliasing

If a refers to an object and you assign `b = a`, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, **it is safer to avoid aliasing when you are working with mutable objects**.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

it almost never makes a difference whether `a` and `b` refer to the same string or not.

9.13 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [4]
>>> print(t3)
[1, 2, 3, 4]
>>> t1 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

9.14 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this:

   ```
   t = t.sort()           # WRONG!
   ```

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at https://docs.python.org/3/library/stdtypes.html#string-methods. The methods and operators that only apply to mutable sequences are documented at https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

```
t.append(x)
t = t + [x]
```

   And these are wrong:

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
orig = t[:]
t.sort()
```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

4. Lists, `split`, and files

   When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the **guardian** pattern when it comes writing programs that read through a file and look for a "needle in the haystack".

   Let's revisit our program that is looking for the day of the week on the from lines of our file:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don't have "From" as the first word as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From': continue
    print(words[2])
```

This looks much simpler and we don't even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From': continue
    print(words[2])
```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails, the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

The error occurs when our program encounters a blank line! Of course there are "zero words" on a blank line. Why didn't we think of that when we were writing the code? When the code looks for the first word (`word[0]`) to check to see if it matches "From", we get an "index out of range" error.

This of course is the perfect place to add some **guardian** code to avoid checking the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we have before we look at the first word:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0: continue
    if words[0] != 'From': continue
    print(words[2])
```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are "interesting" to us and which we want to process some more. A line which has no words is "uninteresting" to us so we skip to the next line. A line which does not have "From" as its first word is uninteresting to us so we skip it.

The program as modified runs successfully, so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, "What might go wrong?"

Exercise 9.1 Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

Exercise 9.2 Rewrite the guardian code in the above example without two `if` statements. Instead, use a compound logical expression using the `and` logical operator with a single `if` statement.

9.15 Glossary

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

element: One of the values in a list (or other sequence); also called items.

equivalent: Having the same value.

index: An integer value that indicates an element in a list.

identical: Being the same object (which implies equivalence).

list: A sequence of values.

list traversal: The sequential accessing of each element in a list.

nested list: A list that is an element of another list.

object: Something a variable can refer to. An object has a type and a value.

reference: The association between a variable and its value.

9.16 Exercises

Exercise 9.3 Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.

Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

Exercise 9.4 Write a program to open the file `romeo.txt`[2] and read it line by line. For each line, split the line into a list of words using the `split` function.

For each word, check to see if the word is already in a list. If the word is not in the list, add it to the list.

When the program completes, sort and print the resulting words in alphabetical order. (Note: this ordering is technically called **lexicographic order** because it is based on the character symbols. Notice that words that start with a capital letter are placed before words that start with a lowercase letter.)

[2] You may download the `romeo.txt` file from `https://www.monead.com/introcompsci/romeo.txt`, which should be stored in the same folder that you are in when you start Python.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 9.5 Write a program to read through the mail box data[3] and when you find line that starts with "From", you will split the line into words using the `split` function. We are interested in who sent the message, which is the second word on the From line.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

You will parse the From line and print out the second word for each From line, then you will also count the number of From (not From:) lines and print out a count at the end.

This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 9.6 Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters "done". Write the program to store the numbers the user enters in a list and use the `max()` and `min()` functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

[3] You may download the `mbox-short.txt` https://www.monead.com/introcompsci/mbox-short.txt, which should be stored in the same folder that you are in when you start Python.

Chapter 10

Dictionaries

A **dictionary** is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

The curly brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `'uno'`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print `eng2sp`, you might be surprised:

```
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2sp['two'])
'dos'
```

The key `'two'` always maps to the value `'dos'` so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2sp['four']}
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a **hash table** that has a remarkable property—the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain why hash functions are so magical, but you can read more about it at https://en.wikipedia.org/wiki/Hash_table.

Exercise 10.1 Write a program that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

10.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

We are effectively computing a **histogram**, which is a statistical term for a set of counters (or frequencies).

The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's the output of the program:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters `'a'` and `'b'` appear once; `'o'` appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

We can use `get` to write our histogram loop more concisely. Because the `get` method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the `if` statement.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

The use of the `get` method to simplify this counting loop ends up being a very commonly used "idiom" in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the `if` statement and `in` operator with the loop using the `get` method. They do exactly the same thing, but one is more succinct.

10.2 Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let's start with a very simple file of words taken from the text of *Romeo and Juliet*.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two `for` loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called **nested loops** because one of the loops is the *outer* loop and the other loop is the *inner* loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating "more quickly" and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the `romeo.txt` file is available at https://www.py4e.com/code3/romeo.txt)

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

10.3 Looping and dictionaries

If you use a dictionary as the sequence in a `for` statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Here's what the output looks like:

```
jan 100
chuck 1
annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

The `for` loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here's what the output looks like:

```
jan 100
annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the `keys` method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

Here's what the output looks like:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

First you see the list of keys in unsorted order that we get from the `keys` method. Then we see the key-value pairs in order from the `for` loop.

10.4 Advanced text parsing

In the above example using the file `romeo.txt`, we made the file as simple as possible by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Since the Python `split` function looks for spaces and treats words as tokens separated by spaces, we would treat the words “soft!” and “soft” as *different* words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat “who” and “Who” as different words with different counts.

We can solve both these problems by using the string methods `lower`, `punctuation`, and `translate`. Using `translate` is the most subtle of the methods.

Using the string translate() method

The `translate` method operates on a string, replacing and/or removing characters. We decide which characters are replaced or removed using another string method called `maketrans`. The method creates a *translation object* which is used by the `translate` method. The operation may sound a little convulted, but we’ll go through a couple of examples to clarify how these methods work together.

You can review Python’s documentation for the `maketrans` method at https://docs.python.org/3/library/stdtypes.html#str.maketrans

The `maketrans` method accepts three string arguments. The first is the set of characters to be replaced. The second is the set of characters to be used as replacements. There is a one-to-one relationship between these strings and they must be the same length. The third argument is a string containing the characters we want to remove from the string.

If we only want to remove characters then we use empty strings for the first two arguments. If we only want to replace characters we leave the third argument out or supply it as an empty string.

Here is an example of using `maketrans` to create a translation object which changes any lowercase ’e’ to ’3’ and any lowercase ’o’ to 0 (zero). It will not remove any characters. Note that we call the method on the **str** class itself.

```
>>> trans = str.maketrans('eo', '30')
```

When we use this translation object on a string, Python will look through the string for a matching character from the first argument. If a match is found then that character’s position will be used to find a matching character in the second argument. That character will replace the original matching one in the string.

You can review Python’s documentation for the `translate` method at https://docs.python.org/3/library/stdtypes.html#str.translate

The following example puts all the translation steps together. We create a string variable named `phrase`. We then create the translation object, called `trans`, in the same fashion as before. Next we use the translate method with our `phrase` string, passing the translation object, `trans`, and assigning the translated string to the variable `newphrase`. Finally we print the two strings on the console to see the effect of the translation.

```
>>> phrase = 'To be or not to be'
>>> trans = str.maketrans('eo', '30')
>>> newphrase = phrase.translate(trans)
>>> print(phrase, '|', newphrase)
To be or not to be | T0 b3 0r n0t t0 b3
```

Let's try one more example. We'll create a translation object that translates all lowercase vowels to uppercase and deletes any digits from a string.

Here is the translation object that does that:

```
alter = str.maketrans('aeiou', 'AEIOU', '0123456789');
```

Remember that the first argument provides the string of characters to match for replacement, the second provides the string of replacement characters, and the third provides a string of characters to be deleted.

Let's use `translate` on a string with our `alter` translation object to see the resulting string.

```
>>> data = 'Temps today will range from 25 to 42'
>>> alter = str.maketrans('aeiou', 'AEIOU', '0123456789');
>>> newdata = data.translate(alter)
>>> print(data, '|', newdata)
Temps today will range from 25 to 42 | TEmps tOdAy wIll rAngE frOm  tO
```

Review the output and make sure you understand why the `newdata` string ends up with those characters. Also, create variations of these examples to check your understanding.

One more thought before we get back to our `romeo.txt` example. We can embed the call to the `maketrans` method inside the call to `translate` and avoid creating a separate variable for the translation object. We'll use that approach in our next program.

Combining split, lower, and translate

We return to our program for processing the `romeo.txt` file. Remember, our intent is to extend our earlier program, removing all the punctuation from the input. Using `maketrans` and `translate` will make it easy to accomplish that.

We'll use the last argument to `maketrans` to list all of the punctuation to be deleted. We will even let Python tell us the list of characters that it considers "punctuation":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

We make the following modifications to our program. Note we place the call to `maketrans` as the parameter to `translate` instead of storing that separately:

```
import string                                         # New Code

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('','',string.punctuation))    # New Code
    line = line.lower()                               # New Code
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)
```

We use `translate` to remove all punctuation and `lower` to force the line to lowercase. Otherwise the program is unchanged.

Part of learning the "Art of Python" or "Thinking Pythonically" is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python **tuples**. We will pick up this example once we learn about tuples.

10.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines.

If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "completely illogical".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

Pretty print the output: Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debugging.

10.6 Glossary

dictionary: A mapping from a set of keys to their corresponding values.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

histogram: A set of counters.

implementation: A way of performing a computation.

item: Another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

key-value pair: The representation of the mapping from a key to a value.

lookup: A dictionary operation that takes a key and finds the corresponding value.

nested loops: When there are one or more loops "inside" of another loop. The inner loop runs to completion each time the outer loop runs once.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value".

10.7 Exercises

Exercise 10.2 Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with "From", then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercise 10.3 Write a program to read through a mail log, build a histogram using a dictionary to count how many messages have come from each email address, and print the dictionary.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Exercise 10.4 Add code to the above program to figure out who has the most messages in the file.

After all the data has been read and the dictionary has been created, look through the dictionary using a maximum loop (see Section 5.8.2) to find who has the most messages and print how many messages the person has.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
```

```
Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 10.5 This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Chapter 11

Tuples

11.1 Tuples are immutable

A tuple[1] is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are **immutable**. Tuples are also **comparable** and **hashable** so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<class 'tuple'>
```

Without the comma Python treats `('a')` as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Another way to construct a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

[1]Fun fact: The word "tuple" comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quituple, sextuple, septuple, etc.

```
>>> t = tuple()
>>> print(t)
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to `tuple` is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

11.2 Comparing tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called **DSU** for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples using the Python built-in `sort`, and

Undecorate by extracting the sorted elements of the sequence.

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
   t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in *reverse* alphabetical order, so "what" appears before "soft" in the following list.

The output of the program is as follows:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

11.3 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables `x` and `y` in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:[2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

A particularly clever application of tuple assignment allows us to **swap** the values of two variables in a single statement:

```
>>> a, b = b, a
```

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

[2]Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as you might expect.

More generally, the right side can be any kind of sequence (string, list, or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

11.4 Dictionaries and tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('a', 10), ('c', 22), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

11.5 Multiple assignment with dictionaries

Combining `items`, tuple assignment, and `for`, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in d.items():
    print(val, key)
```

This loop has two **iteration variables** because `items` returns a list of tuples and `key, val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both `key` and `value` are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the *value* stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is `(value, key)`. The `items` method would give us a list of `(key, value)` tuples—but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

11.6 The most common words

Coming back to our running example of the text from *Romeo and Juliet* Act 2, Scene 2, we can augment our program to use this technique to print the ten most common words in the text as follows[3]:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(line.maketrans('','',string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
    lst.append( (val, key) )

lst.sort(reverse=True)

for val, key in lst[:10] :
    print(val, key)
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out `counts` and ending the program, we construct a list of `(val, key)` tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice `for` loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (`lst[:10]`).

So now the output finally looks like what we want for our word frequency analysis.

```
61 i
42 and
40 romeo
34 to
34 the
```

[3]You may download the `romeo.txt` file from https://www.monead.com/introcompsci/romeo.txt, which should be stored in the same folder that you are in when you start Python.

```
32 thou
32 juliet
30 that
29 my
24 thee
```

The fact that this complex data parsing and analysis can be done with an easy-to-understand 19-line Python program is one reason why Python is a good choice as a language for exploring information.

11.7 Using tuples as keys in dictionaries

Because tuples are **hashable** and lists are not, if we want to create a **composite** key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables `last`, `first`, and `number`, we could write a dictionary assignment statement as follows:

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment in a `for` loop to traverse this dictionary.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

11.8 Sequences: strings, lists, and tuples—Oh My!

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists, and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.

2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. However Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

11.9 Debugging

Lists, dictionaries and tuples are known generically as **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size, or composition, or perhaps you write some code and forget the shape of your data and introduce an error.

For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

11.10 Glossary

comparable: A type where one value can be checked to see if it is greater than, less than, or equal to another value of the same type. Types which are comparable can be put in a list and sorted.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

DSU: Abbreviation of "decorate-sort-undecorate", a pattern that involves building a list of tuples, sorting, and extracting part of the result.

gather: The operation of assembling a variable-length argument tuple.

hashable: A type that has a hash function. Immutable types like integers, floats, and strings are hashable; mutable types like lists and dictionaries are not.

scatter: The operation of treating a sequence as a list of arguments.

shape (of a data structure): A summary of the type, size, and composition of a data structure.

singleton: A list (or other sequence) with a single element.

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

11.11 Exercises

Exercise 11.1 Revise a previous program as follows: Read and parse the "From" lines and pull out the addresses from the line. Count the number of messages from each person using a dictionary.

After all the data has been read, print the person with the most commits by creating a list of (count, email) tuples from the dictionary. Then sort the list in reverse order and print out the person who has the most commits.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 11.2 This program counts the distribution of the hour of the day for each of the messages. You can pull the hour from the "From" line by finding the time string and then splitting that string into parts using the colon character. Once you have accumulated the counts for each hour, print out the counts, one per line, sorted by hour as shown below.

```
Sample Execution:
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercise 11.3 Write a program that reads a file and prints the *letters* in decreasing order of frequency. Your program should convert all the input to lower case and only count the letters a-z. Your program should not count spaces, digits, punctuation, or anything other than the letters a-z. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at `wikipedia.org/wiki/Letter_frequencies`.

Chapter 12

Regular expressions

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been using string methods like `split` and `find` and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful library called **regular expressions** that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions. For more detail on regular expressions, see:

http://en.wikipedia.org/wiki/Regular_expression

https://docs.python.org/3/library/re.html

Many of the example programs in this chapter read the file `mbox-short.txt`. Here are the initial lines from `mbox-short.txt`[1] file to help you picture the strings the programs are reading:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Received: from murder (mail.umich.edu [141.211.14.90])
by frankenstein.mail.umich.edu (Cyrus v2.3.8) with LMTPA;
Message-ID: <200801051412.m05ECIaH010327@nakamura.uits.iupui.edu>
Mime-Version: 1.0
Content-Transfer-Encoding: 7bit
```

[1] You may download the `mbox-short.txt` file from https://www.monead.com/introcompsci/mbox-short.txt, which should be stored in the same folder that you are in when you start Python.

```
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/sakai_2-5-x/content-impl/impl/src/java/org/
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
Author: stephen.marquard@uct.ac.za
Date: 2008-01-05 09:12:07 -0500 (Sat, 05 Jan 2008)
New Revision: 39772

From louis@media.berkeley.edu Fri Jan  4 18:10:48 2008
Return-Path: <postmaster@collab.sakaiproject.org>
...
```

The regular expression library `re` must be imported into your program before you can use it. The simplest use of the regular expression library is the `search()` function. The following program demonstrates a trivial use of the search function.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print(line)
```

We open the file, loop through each line, and use the regular expression `search()` to only print out lines that contain the string "From:". This program does not use the real power of regular expressions, since we could have just as easily used `line.find()` to accomplish the same result.

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is used in regular expressions to match "the beginning" of a line. We could change our program to only match lines where "From:" was at the beginning of the line as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print(line)
```

Now we will only match lines that *start with* the string "From:". This is still a very simple example that we could have done equivalently with the `startswith()` method from the string library. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

12.1 Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period or full stop, which matches any character.

In the following example, the regular expression "F..m:" would match any of the strings "From:", "Fxxm:", "F12m:", or "F!@m:" since the period characters in the regular expression match any character.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line) :
        print(line)
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the "*" or "+" characters in your regular expression. These special characters mean that instead of matching a single character in the search string, they match zero-or-more characters (in the case of the asterisk) or one-or-more of the characters (in the case of the plus sign).

We can further narrow down the lines that we match using a repeated **wild card** character in the following example:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line) :
        print(line)
```

The search string "^From:.+@" will successfully match lines that start with "From:", followed by one or more characters (".+"), followed by an at-sign. So this will match the following line:

From: stephen.marquard @uct.ac.za

You can think of the ".+" wildcard as expanding to match all the characters between the colon character and the at-sign.

From: .+ @

It is good to think of the plus and asterisk characters as "pushy". For example, the following string would match the last at-sign in the string as the ".+" pushes outwards, as shown below:

From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu

It is possible to tell an asterisk or plus sign not to be so "greedy" by adding another character. See the detailed documentation for information on turning off the greedy behavior.

12.2 Extracting data using regular expressions

If we want to extract data from a string in Python we can use the `findall()` method to extract all of the substrings which match a regular expression. Let's use the example of wanting to extract anything that looks like an email address from any line regardless of format. For example, we want to pull the email addresses from each of the following lines:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

We don't want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses `findall()` to find the lines with email addresses in them and extract one or more addresses from each of those lines.

```
import re
s = 'Hello from csev@umich.edu to cwen@iupui.edu about the meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)
```

The `findall()` method searches the string in the second argument and returns a list of all of the strings that look like email addresses. We are using a two-character sequence that matches a non-whitespace character (\S).

The output of the program would be:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-whitespace character. The "\S+" matches as many non-whitespace characters as possible.

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu), but it would not match the string "@2PM" because there are no non-blank characters *before* the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an email address as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0 :
        print(x)
```

We read each line and then extract all the substrings that match our regular expression. Since `findall()` returns a list, we simply check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an email address.

If we run the program on `mbox.txt`[2] we get the following output:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Some of our email addresses have incorrect characters like “<” or “;” at the beginning or end. Let’s declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the “\S” is asking to match the set of “non-whitespace characters”. Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a *single* lowercase letter, uppercase letter, or number “[a-zA-Z0-9]”, followed by zero or more non-blank characters (“\S*”), followed by an at-sign, followed by zero or more non-blank characters (“\S*”), followed by an uppercase or lowercase letter. Note that we switched from “+” to “*” to indicate zero or more non-blank characters since “[a-zA-Z0-9]” is already one non-blank character. Remember that the “*” or “+” applies to the single character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0 :
        print(x)
```

[2]You may download the `mbox.txt` file from https://www.monead.com/introcompsci/mbox.txt, which should be stored in the same folder that you are in when you start Python. This file is similar to mbox-short.txt but has many more lines of text in it.

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Notice that on the "source@collab.sakaiproject.org" lines, our regular expression eliminated two letters at the end of the string (">;"). This is because when we append "[a-zA-Z]" to the end of our regular expression, we are demanding that whatever string the regular expression parser finds must end with a letter. So when it sees the ">" after "sakaiproject.org>;" it simply stops at the last "matching" letter it found (i.e., the "g" was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

12.3 Combining searching and extracting

If we want to find numbers on lines that start with the string "X-" such as:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

we don't just want any floating-point numbers from any lines. We only want to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

```
^X-.*: [0-9.]+
```

Translating this, we are saying, we want lines that start with "X-", followed by zero or more characters (".*"), followed by a colon (":") and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period "[0-9.]+". Note that inside the square brackets, the period matches an actual period (i.e., it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print(line)
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

But now we have to solve the problem of extracting the numbers. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression, they are ignored when matching the string. But when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0 :
        print(x)
```

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating-point number to indicate we only want `findall()` to give us back the floating-point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

The numbers are still in a list and need to be converted from strings to floating point, but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)
```

Translating our regular expression, we are looking for lines that start with "Details:", followed by any number of characters (".*"), followed by "rev=", and then by one or more digits. We want to find lines that match the entire expression but we only want to extract the integer number at the end of the line, so we surround "[0-9]+" with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```

Remember that the "[0-9]+" is "greedy" and it tries to make as large a string of digits as possible before extracting those digits. This "greedy" behavior is why we get all five digits for each number. The regular expression library expands in both directions until it encounters a non-digit, or the beginning or the end of a line.

Now we can use regular expressions to redo an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

and wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this in a far simpler way with the following regular expression:

```
^From .* [0-9][0-9]:
```

The translation of this regular expression is that we are looking for lines that start with "From " (note the space), followed by any number of characters (".*"), followed by a space, followed by two digits "[0-9][0-9]", followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

```
^From .* ([0-9][0-9]):
```

This results in the following program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0 : print(x)
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
...
```

12.4 Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are "normal" and we want to match the actual character such as a dollar sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar sign with a backslash, it actually matches the dollar sign in the input string instead of matching the "end of line", and the rest of the regular expression matches one or more digits or the period character. *Note:* Inside square brackets, characters are not "special". So when we say "[0-9.]", it really means digits or a period. Outside of square brackets, a period is the "wild-card" character and matches any character. Inside square brackets, the period is a period.

12.5 Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings with special characters in them that communicate your wishes to the regular expression system as to what defines "matching" and what is extracted from the matched strings. Here are some of those special characters and character sequences:

^
Matches the beginning of the line.

$
Matches the end of the line.

.
Matches any character (a wildcard).

\s
Matches a whitespace character.

\S
Matches a non-whitespace character (opposite of \s).

*
Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s).

*?
Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s) in "non-greedy mode".

+
Applies to the immediately preceding character and indicates to match one or more of the preceding character(s).

+?
Applies to the immediately preceding character and indicates to match one or more of the preceding character(s) in "non-greedy mode".

[aeiou]
Matches a single character as long as that character is in the specified set. In this example, it would match "a", "e", "i", "o", or "u", but no other characters.

[a-z0-9]
You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.

[^A-Za-z]
When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an uppercase or lowercase letter.

( )
When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

\b
Matches the empty string, but only at the start or end of a word.

\B
Matches the empty string, but not at the start or end of a word.

\d
Matches any decimal digit; equivalent to the set [0-9].

\D
Matches any non-digit character; equivalent to the set [^0-9].

12.6 Bonus section for Unix users

Support for searching files using regular expressions was built into the Unix operating system since the 1960s and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into Unix called **grep** (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command-line window.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

This tells `grep` to show you lines that start with the string "From:" in the file `mbox-short.txt`. If you experiment with the `grep` command a bit and read the documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character "\S" so you will need to use the slightly more complex set notation "[^]", which simply means match a character that is anything other than a space.

12.7 Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

WWelcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the functions in the module as follows:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

You can also get a small amount of documentation on a particular function using the dir command.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

The built-in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

12.8 Glossary

brittle code: Code that works when the input data is in a particular format but is prone to breakage if there is some deviation from the correct format. We call this "brittle code" because it is easily broken.

greedy matching: The notion that the "+" and "*" characters in a regular expression expand outward to match the largest possible string.

grep: A command available in most Unix systems that searches through text files looking for lines that match regular expressions. The command name stands for "Generalized Regular Expression Parser".

regular expression: A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.

wild card: A special character that matches any character. In regular expressions the wild-card character is the period.

12.9 Exercises

Exercise 12.1 Write a simple program to simulate the operation of the `grep` command on Unix. Ask the user to enter a regular expression and count the number of lines in the `mbox.txt`[3] file that match the regular expression entered by the user:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Exercise 12.2 Write a program which asks the user for a file name and then reads that file, looking for lines of the form:

```
New Revision: 39772
```

(e.g. lines beginning with the text `New Revision:` followed by a number) and extract the number from each of the lines using a regular expression and the `findall()` method. Compute the average of the numbers and print out the average.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```

[3] You may download the `mbox.txt` file from https://www.monead.com/introcompsci/mbox.txt, which should be stored in the same folder that you are in when you start Python.

Chapter 13

Exploring Computer Science

13.1 What is Computer Science

Computer Science (CS) is a term used to describe a vast field. Our everyday exposure to CS may leave us with a narrow view of computing. We often associate computing with a few types of hardware, such as cell phones and laptop computers. However, the field of CS goes far beyond these everyday devices and the programs we often use to play games, interact on social media, or make online purchases.

Merriam-Webster defines *computer science* as ”a branch of science that deals with the theory of computation or the design of computers” (*Computer science: Definition and meaning* n.d.). The theory of computation focuses on problems that can be solved using computers and, in more general terms, involves creating models. We use models to help us understand concepts and make predictions about the world around us. You have probably used a globe, which is a physical model of the Earth and can be used to solve problems such as what direction you should travel to get from one place to another or the distance between those locations. In computer science, the device housing the model is a computer, but the intent is the same, to answer questions about the real-world concept we are modeling. From this perspective, computer science is a field involving understanding concepts and finding ways to express them in a model that a computing device can use.

Before we delve into the breadth of CS as a profession, let us consider its impact on our lives.

13.2 Impact of CS on Society

Author’s Note: This section, describing the impact of CS on society, is taken from Read (2023).

Computer-based systems are woven into the fabric of society, touching many aspects of people's daily lives (Koya and Chowdhury 2020). At a macro level, computers manage utilities, such as electricity, water, and phone services, integrating large and distributed data sets (Koya and Chowdhury 2020). Airlines, cargo ships, emergency services, drivers, and pedestrians rely heavily on the Global Positioning Service (GPS), whether looking for a route to a distant location or finding a nearby convenience store (McNeff 2002). Medical professionals rely on computer systems to diagnose and treat patients (Cirillo et al. 2020). Computers and the Internet have displaced broadcast media as the primary news and entertainment source, significantly altering how people consume information (Chalaby 2016).

The universal presence of computers in society leads to those systems significantly impacting life and culture. Consider three powerful computer-based platforms that dramatically changed society: the Internet, cell phones, and artificial intelligence (AI). Each has fundamentally changed how people access news and information, how individuals and groups communicate, how organizations develop and market products, and how governments monitor their citizens.

The Internet provides an open platform for connecting information (Berners-Lee et al. 2010). It manifests an egalitarian vision of "anyone can say anything about anything" (*Resource description framework (RDF): Concepts and abstract syntax* n.d.). Without an effective way to assess the provenance and trustworthiness of sources, the reliability of information obtained from the web becomes suspect. Society has seen this impact through the growing prevalence of misinformation and loss of privacy (Berners-Lee 2014; Zhou and Zafarani 2020).

Cell phones provide a constant connection to people and information. The connectivity is two-way, providing governments and private industry information about a person's behaviors and whereabouts (Dienlin and Trepte 2015). Consumers appreciate receiving a coupon for a store they happen to be walking past or the ease of asking Siri to find the nearest Starbucks. However, the amount of electronic data collected and aggregated about people interacting with their phones is enormous (Dienlin and Trepte 2015). People may become addicted to their cell phones, impacting their mental state, ability to focus, and feelings of connectedness to others (De-Sola Gutiérrez, Fonseca, and Rubio 2016). The unchecked integration of these portable, always-on computers into everyday life has transformed how people and organizations interact.

In tandem with the large amounts of data now available to corporations and individuals, using AI to identify information quickly has grown in popularity (Vinichenko et al. 2021). AI builds upon the field of machine learning, "a branch of computer science that broadly aims to enable computers to 'learn' without being directly programmed" (Bi et al. 2019, p. 2222). This technology allows companies to find patterns and similarities between people, which may lead to identifying a health condition or offering a product they are likely to buy (Fennelly 2021; Ma and Sun 2020). Because the techniques used to develop AI-based systems use society-generated data, those systems risk reproducing and amplifying biases carried by the individuals or created by the society in which they operate (Wiens, W. Nicholson Price, and Sjoding 2020). The automated creation of computer systems driven by AI creates many opportunities to help or harm society.

Each of these technologies continues to become ever more ingrained in people's daily lives and the operations of governments and corporations. These, and new CS-related fields, such as quantum computing and blockchain databases, will continue to increase the prevalence of computing systems in the fundamental operation of society.

13.3 The Information Economy

While manufacturing drove the economy through much of the late 19th and early-to-mid 20th century, a manufacturing economy, a shift has occurred during the last 40 years where companies derive value from information processing. The term "information economy" describes this shift, which continues to impact people's careers. Many jobs require people to manipulate data, making the skills to quickly reformat, summarize, or create models from data valuable whether or not a job is considered an information or computer technology position.

As the use of data becomes more prevalent in many jobs, the type of work changes. For example, while manufacturing in the early 20th century involved mostly human-operated machines, now robots are often utilized to run the manufacturing processes, with computers and information controlling them. Instead of operating levers on a drill press, a worker uses a computer to manipulate values that control the press.

The information-centric nature across industries can be seen in the automation of heating and ventilation systems (smart thermostats), robotic surgery, and self-driving trucks, to name a few recent examples. Households and individuals are becoming more connected using health trackers like Fitbit, smart light bulbs, and doorbell cameras. Many of these devices work in an "always connected" mode, made possible by Internet access and WiFi availability.

The bottom line: *an understanding of how computer-based systems operate empowers an individual to innovate in the information economy.*

13.4 CS Jobs and Careers

The breadth of jobs connected to CS is wide and the types of work varied. Unfortunately, many people conflate CS with programming. The stereotype of a CS professional isolated hour after hour, rarely interacting with others, staring at a computer screen as they type source code into an editor, is prevalent yet inaccurate. Part of the issue is how students are introduced to CS and part is due to the diverse nature of CS work.

At its heart, CS in about information and computation. These are broad terms and encompass the fundamentals of how our universe operates. Physics, chemistry, biology, music, dance, language, and any other topic or concept you can imagine relates to information and computation. Whether quarks, atoms, cells, biological systems, planets, or asteroids, all are information and all interact in ways that involve computation. In CS we seek to represent information as data and interactions as programs.

Thinking about CS in this way may feel overwhelming, but just like studying any topic, we take things step by step. You won't become a chemical engineer in a day, similarly it takes time to understand many important CS concepts. This leads us to part of the reason people may view CS as programming; it tends to be the first topic students are introduced to in CS. This book has many chapters that introduce you to programming, algorithms, and data structures; in this chapter we'll explore other aspects of CS. The goal of this chapter is to help you glimpse the breadth of the CS field.

We'll begin with an overview of CS careers. Later we'll consider professional roles and the intersection of those roles within specific industries.

13.4.1 CS Career Paths

An obvious career path decision in most specialties is whether to work in private industry, academia, or the public sector. In some ways, this is a false choice. Academia is often associated with research and teaching, private industry evokes thoughts of creating products or services, and working in the public sector often aligns with implementing and managing social programs. However, academic research informs industry practice, corporations engage in research and development, and governments invest significant resources in academia and private industry to advance new ideas. The fundamental distinction is one of an individual's interests, are they motivated to teach, build products, conduct multi-disciplinary research, work for social change, or apply some combination of technology-related interests?

A career choice brings with it experience and educational requirements. Again, this is not unique to CS. The more specialized a role, the more hands-on experience, education, or a combination of the two will be required. An excellent way to explore careers and plan your journey is to learn about a field and the different types of jobs it offers and talk to people who do that work to understand how they advance their careers.

To highlight the variety of work opportunities that CS offers, the next sections consider CS research followed by CS professional roles prevalent in private and public sectors.

13.4.2 CS Research Areas

CS researchers advance our understanding of many specific concepts related to information and computation. Advancements made through research are applied by improving current applications or enabling new solutions to be created. Table 13.1 defines several active areas of CS-related research.

Table 13.1: CS-related Research Areas

Area	**Description**
Algorithm and data structure development	An algorithm is a set of steps followed to complete a process. A data structure is a way to represent information in the computer. Both are fundamental to the operation of every program, determining the resources required for the program to work. Those resources, such as memory consumption and computing effort, directly affect the user's experience, including time for the program to respond, battery consumption, and network utilization. Finding a more efficient way for a program to meet a user's needs has tangible benefits that reduce run time and costs. The continued progress in quantum computing puts further focus on this area of research, as these new computing paradigms will require novel algorithms.
Automata theory	Automata theory considers abstract machines and the problems that can be solved using them. This field relates to the theory of computation and delves into languages and state machines.
Cryptography	Cryptography is a broad field concerned with protecting information from corruption, loss, and theft. Much of the work relies on mathematical and algorithmic designs. The area continues to be an active one for researchers as computers become more powerful, including the arrival of quantum computers, requiring new cryptographic approaches to provide information security.
Database theory	This field focuses on data storage and retrieval. The complexity of systems requires databases to support a high level of concurrency, maintain consistency across updates, and allow for powerful searching capabilities. In this field, researchers define data structures and algorithms to address shortcomings in database solutions as the interconnected nature of computing systems continues to grow.

Area	Description
Information theory	Information theory considers the representation of information. It is central to cryptography, databases, and machine learning. Finding ways to represent information more efficiently impacts data access performance across network and storage devices. More efficient information representation can improve a user's experience, such as faster retrieval of search results or higher quality video resolution at a lower cost over a broadband connection.
Programming language theory	Programming languages serve as the layer between a person's understanding of the program they want to create and the computer's view of the logic. Thousands of programming languages and several significant language paradigms are in use today. Researchers continue to create new programming languages, and this effort will accelerate as specialized languages for concurrent and quantum-based computing are developed.

13.4.3 CS Professional Roles

At the outset of this chapter, we discussed that the application of CS to build products involves creating models that a computer can use. If you think about all the concepts that make up the world that one might choose to model, you will begin to understand why the careers in CS vary widely. At the outset, there are two dimensions: defining a model, such as the information necessary to express it, and the field of interest, such as farming, medicine, or manufacturing. The first dimension, defining the model, has led organizations to identify CS-related roles and require a team of people who understand CS concepts. The second, specialization in a field, requires individuals with expertise, such as farmers, doctors, and machinists.

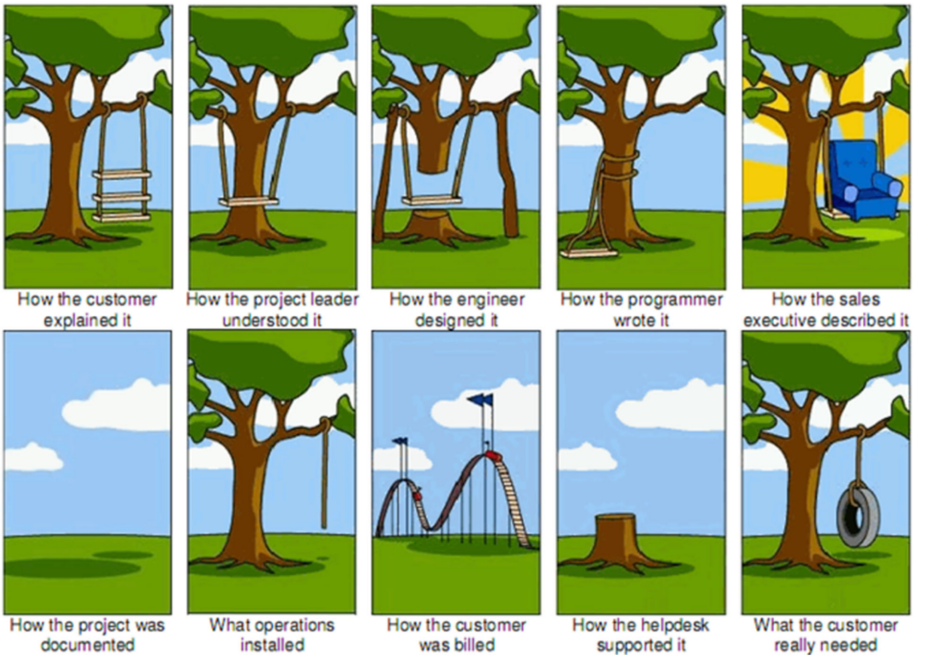
The goal of many technology projects is a computer-based solution, often referred to as a product. The computer may be visible to the user, with a screen and keyboard used to interact with it. In other cases, the human-machine interface may obscure the fact that a product incorporates a computer, such as the controls for an aircraft or recording studio mixing board.

Computer-based products come in many forms, such as mobile phone apps, medical equipment, and websites. An organization utilizes many people when designing, building, testing, and ultimately deploying a product. In many cases, CS-team members work across multiple project teams, where they are brought into the team to share their expertise as needed.

A lot can go wrong in the journey from an idea to the final product, especially when more than one person is involved. Communication across the team is vital to achieving success; again, this is not unique to CS. Effective communication is a vital component for success in any field.

Figure 13.1 is a facetious example depicting a project that gets off-track as different team members share their understanding and ideas. Although the example is humorous, it reminds us that there are many potential failure points as a team works to design and build a product. As we discuss the various roles in a computer-based project, consider which ones you believe are crucial to ensuring the final product matches the original intent.

Figure 13.1: A Project Gone Awry



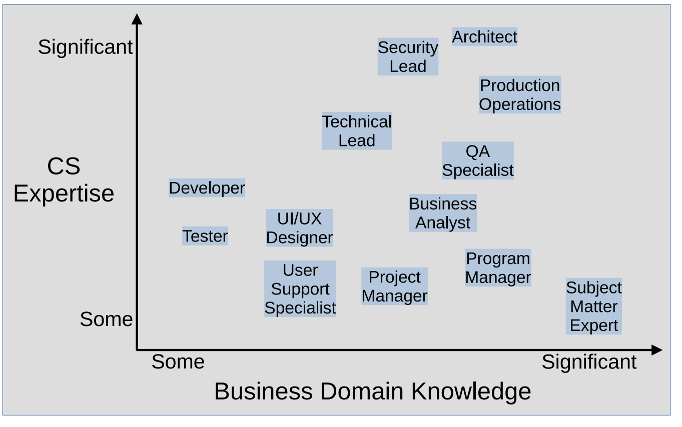
Job and Role Titles

There is no consensus across private or public industries regarding the titles used for CS job roles. Terms such as developer, programmer, software engineer, designer, analyst, and architect, are often used as the titles for CS job postings. However, the job duties and desired skills listed for these positions will vary widely. When considering CS job opportunities, it is best to look past the position title and use the skills and responsibilities to understand what the job entails.

Common CS Project Roles

Figure 13.2 depicts many roles involved in creating a CS-based product, comparing how much business versus CS expertise a person needs for the role. These relative comparisons are not absolutes, and different team members will have a mix of skills. Table 13.2 describes the roles depicted in Figure 13.2. The arrangement of the table does not represent a management reporting structure. Often, a person's manager is not someone they work with daily; instead, they work with their product team(s), and their manager is there to provide support within the organization, such as career planning, advancement, and training.

Figure 13.2: Product Team Roles Mapped to CS and Domain Knowledge



CS Team Roles: CS knowledge, domain knowledge

Table 13.2: CS-related Team Roles

Role	Description
Program Manager	Manages multiple products. Assures teams have the resources and people they need, clarifies to team members how the products fit into the organization's goals and objectives and coordinates requirements and schedules with internal and external stakeholders.
Product Manager	Manages the creation and maintenance of a product. Coordinates schedules across team members, assures team has the resources and people needed to succeed, clarifies ambiguities in priorities for features, communicates status information to stakeholders.

Role	Description
UI/UX Designer	Responsible for user experience and user interface design. They assess how people will interact with the product, including considering accessibility, such as how those who are visually or auditorially impaired, color blind, or unable to control an interface with a mouse or standard keyboard will use the product. They also consider the application's aesthetics, using colors, fonts, and positions of controls, physical and on-screen, to create a pleasant and coherent interface for the user.
Architect	Designs the infrastructure for the product. Most computer-based products involve multiple types of technology that have to be linked together. These technologies, such as databases, web services, user interfaces, authentication and authorization (login and permissions), and document management (images, videos, audio), can be used differently. The architect defines how the product teams will use these technologies. They also ensure the products continue working well when user load increases, known as load balancing, or when a piece of hardware fails, known as failover.
Security Lead	Responsible for the security aspects of the product. Protecting user data requires careful design of the product's hardware and software. This person understands the threats to the product and its data and works with the team to design mitigations to thwart attackers. Security is a broad area in an organization, including physical and virtual security, and many companies have security teams with individuals responsible for identifying and handling different types of breaches.
Technical Lead	Responsible for the overall technical design of the product. The role is sometimes referred to as the design lead. These team members manage decisions about programming languages, third-party packages, data structures, algorithms, and test integrations. This individual has worked as a developer and gained experience with multiple programming languages, packages, development tools, and platforms.

Role	Description
Business Analyst	The business analyst (BA) is responsible for understanding and communicating the requirements for the product. This individual is the bridge between the business and computer science experts. BAs must understand the business objectives for the product and have some experience with CS concepts, such as basic programming, data structures, and testing. BAs are the "voice of the customer," assuring the team remains focused on creating a product that meets the organization's and users' needs.
Developer	Responsible for writing the programs required for the product. Developers have expertise in one or more programming languages and third-party packages or libraries. Developers may specialize in back-end, mid-tier, front-end, and full-stack development. The back-end involves accessing and storing data, including securing the information. Mid-tier development focuses on sending information between systems, often as web services. Front-end developers focus on the user interface, whether using graphical development environments for smartphones and computers or web technologies such as HTML and CSS for web browsers. Full-stack developers have experience and expertise throughout all application tiers.
Tester	Responsible for unit and integration testing of the programs used in the product. The individual is familiar with testing and mocking frameworks and performance and load-testing tools. They play a vital role in assuring that the code created by the developer does what it is supposed to do and does not do things it should not, such as failing to properly handle an unexpected input value.

Role	Description
QA Specialist	The quality assurance (QA) team is responsible for verifying the entire product's quality. This verification requires testing the completed product to be sure it does everything the product owner requested and that the BAs documented. It also means verifying that the product comprehensively protects users and their information from harm or misuse, such as loss or corruption of data. The QA function in an organization often operates in an environment that matches that of the final production environment but is specifically designed to record, playback, and verify the operations of the product using automation tools. QA is also responsible for verifying that the product will work under anticipated conditions, such as a large number of users using the product simultaneously or being able to failover to a backup site if a natural disaster strikes.
Production Operations	The production team is responsible for the ongoing operation of the product once QA has approved it. The production team manages the organization's infrastructure, including servers, networks, backup power supplies, physical security of equipment, user accounts, and Internet access. The team includes people knowledgeable in operating systems, networking protocols, application configuration, database management, physical security, and help desk services. These individuals keep systems functioning and help users when problems arise.
Subject Matter Expert	A subject matter expert (SME) is an expert in the business or field that makes up the focus of the product. For healthcare companies, SMEs are doctors and nurses; at legal companies, they are lawyers and paralegals; for tax and finance companies, they are accountants and banking experts. An SME provides detailed information about how the product must work and can answer questions about its features and operations. The SME, BA, and QA often work closely to ensure the final product meets users' needs.

Role	Description
User Support Specialist	User support specialists have several responsibilities. They create documentation for users, train users, and provide scripts for the help desk to use when a problem arises. In this role, the individual works with the BA, development team, and QA, to ensure documentation is accurate and meets the user's needs. In their role as trainers, they are often the face of the product for users, becoming a trusted resource when a user has additional questions.

After considering all these roles, you may see that computer science is a broad field with opportunities for people of many different backgrounds to combine their talents to create unique products. Whether a new smartphone game app or an advanced surgical robot, CS expertise is a critical component many organizations need to create their products. What makes CS interesting is that as someone gains experience, they can choose different career paths, some preferring to become more technical and others finding more reward in business-centric and user-focused roles.

An important point to note is that programming, primarily the developer's domain, represents a small part of the team and overall effort of implementing a computer-based product. People often equate CS and programming, believing there is little else to CS than writing code; however, that is not the case. Programming is integral to creating a computer-based product, but it only represents some of the work.

13.5 A Survey of CS-related Careers

We have provided this section to give you an idea of the breadth of work that intersects with CS. It is a survey touching on a few examples across different specialties, and the descriptions are purposefully brief. You can always follow up with your instructor or do online research if a role sounds interesting.

13.5.1 Arts and Media

Figure 13.3: Video Editing

Table 13.3: Arts and Media Technology Careers

Role	Description
Film and video editing	Editors work with directors and producers to pull together all aspects, such as video, animation, special effects, sound effects, soundtracks, and dialog, of the movie or show. Editors use specialized hardware and software to complete the editing work.
Multimedia art and animation	Whether special effects in movies or the virtual worlds of immersive games, multimedia artists use their artistic skills and knowledge of animation and drawing software to bring their ideas to life.
Video game design	Video game design overlaps with movie making. Games need plots, game elements, animation, dialog, soundtracks, and much more. Game designers combine these elements using different software packages and libraries to create content and implement the game's logic.
Web design	Developing appealing, useful, and responsive websites is a creative endeavor bringing together visual and technical design. There are many frameworks and languages that website developers choose from based on their needs, time frame, and budget.

13.5.2 Data

Figure 13.4: Data Analysis

Table 13.4: Data Careers

Role	Description
Data analysis	Involves summarizing and connecting the data in ways that help business leaders identify risks and opportunities in their operations and markets. The work often involves finding novel visualizations to make the information apparent.
Data science	This is a broad term for work that involves manipulating, summarizing, or identifying patterns in data. Data science includes data mining, machine learning, and AI.
Data wrangling	Although data analysis and data science get much attention, these roles need high-quality, curated data. Those adept at data wrangling manipulate data from different sources, identify and fix quality issues, and combine related information to create unbiased and well-defined data sets for analysis and mining.
Database administration	The volume, velocity, veracity, and variability in data make its management complex. At the broadest level, digital data exists in structured and unstructured forms. Database administration involves the storage, retrieval, organization, and protection of a constantly-growing collection of data, making sure it is available for efficient access, but only to those authorized to access it.

13.5.3 Data Security

Figure 13.5: Blue team

Table 13.5: Data Security Careers

Role	**Description**
Blue team	This team identifies, stops, and mitigates the efforts of people attacking a company's systems. This team develops plans for handling attacks, works with law enforcement if a breach occurs, and takes the lead in handling active attacks. They are the flip side of the Red Team (see: Red team).
Honeypot administration	A honeypot is a computer with no business purpose other than to attract and detect attackers. Those operating honeypots seek to make the system look like a real business system and then carefully record the actions of those who access it, knowing they are attackers. Honeypots are a powerful tool for finding insider and outsider threats to an organization.
Pen testing	Companies hire pen testers to breach their organization's systems. This work might include getting past physical security barriers and breaking into computers over a network. Pen testing is a popular way for companies to assess the quality of their cyber defenses.
Red team	This team is an adversary to the Blue team (see: Blue team). This team takes actions that an attacker would take to identify weaknesses in defenses and allow the Blue team to test their procedures. Blue and red team members often rotate between teams to broaden their security skills.

13.5.4 Education

Figure 13.6: Educational Software

Table 13.6: Educational Technology Careers

Role	Description
Content creation	Online education, and flipped classrooms require digital educational materials. Content creators are knowledgeable in their fields, whether cooking, biology, music or many other subjects. They also consider how to keep online students engaged using gamification techniques, well-designed graphics, and interactive exercises. Teams work to create an engaging, thought-provoking, and immersive learning environment that adjusts to the learner's needs.
Educational software design	To deliver educational content online effectively, specialized software is needed. The programs include platforms for learning, document, and workflow management. These tools are often developed as web-based environments with additional features for mobile devices.
Teaching and training	The foundation of education are teachers and students. As students become more tech-savvy, teachers' platforms and approaches must keep pace. Finding ways to combine effective pedagogical and andragogical techniques with technology can create a more rewarding experience for the instructor and lead to more effective learning by the student. Teaching is not limited to schools; many businesses have trainers who train employees on internal tools and procedures or train customers to teach them how to use the company's products.

13.5.5 Finance and Insurance

Figure 13.7: Risk Analysis

Table 13.7: Finance Technology Careers

Role	Description
Credit and portfolio analysis	Companies of all kinds utilize many data-centric metrics to assess their financial health and identify issues they must address. Producing accurate and timely metrics requires significant data processing, statistical, and financial expertise. Organizations define Key Performance Indicators (KPIs) to monitor their operations, each requiring the integration and synthesis of data.
Investment banking	Identifying good investments, and keeping track of investment performance are vital for many financial institutions. Larger institutions coordinating significant financial events, such as mergers and acquisitions, rely on data analysis and modeling to recognize good investment opportunities.
Financial planning	Working with individuals and organizations to plan for future financial needs is rewarding. Financial planners utilize a variety of data sources and data modeling tools, coordinating with the goals of each customer to identify optimal investment strategies.
Risk analysis	Companies assess risk for many reasons. For example, insurance companies make their money by properly modeling the financial risk presented by the insured. Insurance exists in many domains, such as healthcare, property and casualty, automobile, and insurance company portfolios. Utilizing data analysis, modeling, and machine learning, insurance companies design their offerings to be affordable and ensure they can pay the claims for losses suffered by their customers.

13.5.6 Government

Figure 13.8: Cyber Terrorism Defense

Table 13.8: Government Technology Careers

Role	Description
Cryptographic analysis	Cryptography is an active area of research. It serves as the basis for securing data stored in a system or accessed across the Internet. Attackers and ever more powerful computers consistently drive the need for new and more effective ways to secure information. This field involves math and creativity. There are no formal proofs of a cryptographically secure algorithm, and careers in cryptography range from creating new cryptographic approaches to finding weaknesses in existing or proposed algorithms.
Cyber terrorism defense	Governments and companies find themselves targeted by cyber attackers and terrorists. To protect themselves and those who use their services, they maintain teams of cyber experts to defend their systems. These include blue teams (see: Data Security: Blue team), those with military experience, and others with deep knowledge of application and network protocols.
Vulnerability discovery	Similar to pen testing (see: Data Security: Pen tester), teams working in vulnerability discovery seek to identify and remediate weaknesses in an organization's infrastructure. These teams often work with software development teams and guide them to strengthen the security of the application the team is building. They also work with QA to design effective security-focused tests. In some cases, governments will scan and proactively alert companies to vulnerabilities to thwart attackers.

13.5.7 Healthcare

Figure 13.9: Clinical Informatics

Table 13.9: Healthcare Technology Careers

Role	**Description**
Clinical informatics	Takes data from medical devices, patient record systems, pharmaceutical databases, and genetics research to identify and treat health issues. Teams involve experts in all aspects of healthcare and computer science. Large datasets are combined, analyzed, modeled, and assessed. These roles exist in healthcare companies, hospitals, doctors' offices, research institutions, and companies that serve those fields.
Outcomes analysis	A vital part of improving healthcare is identifying good versus poor treatment outcomes. The work can also involve looking at recent trends to identify changes to environments or other factors that require adjusting treatment protocols quickly.
Medical coding and billing	Healthcare payers (insurance companies) utilize healthcare data in interesting ways. One area of particular interest is identifying healthcare fraud, which does more harm than just wasting money; it diverts funding from those who need it. Using data analysis and machine learning, billing teams find erroneous or unusual diagnoses or treatment patterns.
Pharmacy benefits administration	Assuring patients get the drugs they need without undue risk due to drug interactions involves the integration of patient databases, pharmaceutical data, genetic data, and insurance records. Identifying dangerous drug interactions goes beyond mapping drugs that interact to consider prior diseases the patient has had and their genetic makeup.

13.5.8 Logistics/Transportation

Figure 13.10: Supply Chain Management

Table 13.10: Logistics Technology Careers

Role	**Description**
Blockchain integration	Blockchain technology allows companies to share a database whose transactions are secured using cryptography. Although often conflated with cryptocurrencies, blockchain is an approach to a decentralized database that companies can use to track the supply and movement of goods and services. Those with blockchain expertise often work with product development teams to integrate a blockchain into the platform.
Self-driving vehicles	The development of autonomous vehicles rely on AI techniques, including machine learning. It is infeasible to program a car or truck manually; it must be able to recognize, respond, and adapt to different conditions. Due to an ongoing shortage of qualified drivers, the trucking industry is actively pursuing self-driving trucks, particularly to handle the ”long haul” (highway) aspects of trucking.
Supply chain management	Coordinating the production and movement of goods in a global economy involve processing data from many different companies, including manufacturing, shipping, and retail. Each company has unique systems and data formats, complicating things further. Skills in data management, data wrangling, and data analysis are consistently in demand, as are team members for companies building applications that coordinate supply chains between organizations.

13.5.9 Machine Learning

Figure 13.11: Virtual Reality

Table 13.11: Machine Learning Careers

Role	Description
Artificial intelligence	AI is a catch-all term for systems that demonstrate intelligence, meaning they can solve problems independently. Much of AI currently relies on machine learning, but AI is broader than that. Numerous roles are related to AI, including algorithmic research, computer hardware, specialized sensors, and software. AI can automate many existing technologies, such as robots, online support centers, and diagnostic services.
Data mining	This field is used in most industries to identify opportunities from their data. Unlike many other uses of machine learning, data miners may not know what they are looking for; rather, they explore the data to see if there is a model to find.
Operations research	Those working in this field use data analytics and data mining to model different scenarios. Their goal is to identify the optimum strategies or solutions to problems posed by an organization. Many organizations have teams providing this service, including government departments and larger corporations.
Virtual reality (VR)	The VR field has gone from a gamer-focused technology to broader applications, including psychological research and treatment, architecture and construction, and medical training. VR teams consist of SMEs in different fields, such as psychology, and those skilled in software development and the visual arts.

13.5.10 Manufacturing and Utilities

Figure 13.12: Robotics

Table 13.12: Manufacturing and Utilities Technology Careers

Role	Description
Enterprise resource planning	Enterprise resource planning (ERP) is a complex task that combines data from engineers, industrial designers, manufacturing SMEs, robotic systems, customers, and suppliers to map out the scheduling of production work across a factory. The popularity of just-in-time (JIT) manufacturing adds to the complexity of the effort, which involves data analysis, specialized planning, and operating software.
Industrial automation	Utility and infrastructure organizations, such as power generation and subways, utilize specialized computer hardware and software, known as supervisory control and data acquisition (SCADA) systems, to monitor and control their equipment. Many organizations connect SCADA systems to the Internet for remote access, yet these systems were designed decades ago, predating computer networking with no consideration of data security. There is widespread concern about attackers disabling power or transit systems through Internet attacks leading to increased demand for people with expertise in configuring and protecting these systems.
Robotics	Many aspects of manufacturing involve automated processes. These systems require specific instructions and data to perform the correct actions at the right time. As companies fabricate new items or change the materials used to build a part, the systems managing the process need to change. This field utilizes a wide variety of specialized hardware and software.

13.5.11 Music and Entertainment

Figure 13.13: Editing and Mixing

Table 13.13: Music and Performance Technology Careers

Role	Description
Arranging	Often, an artist works with an arranger to take the music they have written and work out the instrumentation used in its performance. Music notation and musical instrument digital interface (MIDI) systems play a prominent role in this work. Artists and arrangers can use musical notation on a computer to control digital instruments and hear the result immediately. Even if the final performance will be on non-digital instruments, this approach speeds the process.
Engineering, recording, and mixing	Music recording involves combining and balancing many input sources. These inputs are the sounds from singers and instrumentalists, stored as computer data. Sound engineers shape the tonal experience for the listener, sometimes bringing together hundreds of individual sound sources. Engineers use soundboards that match the ergonomics of analog equipment, but control computer systems that mix and store the audio data.
Performance	Many artists include digital instruments in their ensemble and use MIDI to run aspects of their performance and coordinate an array of special effects. Far beyond the music itself, integrating musical instruments and other aspects of a show, such as lighting and specialized video, requires the work of visual artists, audio engineers, electricians, and developers with expertise in programming the specialized systems used to interface and coordinate data between these devices.

13.5.12 Research

Figure 13.14: Agriculture Modeling and Management

Table 13.14: Research Technology Careers

Role	**Description**
Medical device research	The pace of change in medicine continues to accelerate. Nanotechnology, wearables, robotic surgery, and WiFi-enabled implants offer an expansive world of possibilities for detecting and treating many health conditions. Teams include doctors, nurses, hardware specialists, data analysis, and software development teams to develop, test, and operationalize these devices. The work is not limited to the device's functions; teams create applications to control, record, analyze, and respond to the device's output. Further, specialized training for medical personnel and patients must be created and presented. Being healthcare-related, teams must account for security and privacy to prevent unintended changes to the device's operation and to protect patient privacy.
Modeling	Modeling provides tools to evaluate different options, test multiple hypotheses, and seek optimum solutions in many fields, such as agriculture, biodiversity, climate change, ecology, water management, and waste management. There are many opportunities to combine the expert knowledge of SMEs and information from satellites, weather stations, and specialized sensors, using data science techniques to understand and begin to manage significant challenges in our world.

13.6 Exercises

Exercise 13.1 Look up Moore's Law and explain the concept Moore described and its implications on computers and other electronic devices. Where have you seen evidence of this?

Exercise 13.2 Look up the term "information economy." Define and explain it in your own words. Consider a career or field that interests you and find or suggest ways computers impact or might impact the work done in that field.

Exercise 13.3 Create a table listing all the computing devices and programs (software) you have come into contact with in the last month, their purposes, and the total time you have interacted with them during the month. Expand your list to include other computing devices and programs and their purposes; these might be devices or software you have come into contact with in the past or that you are aware of in other ways (e.g., heard about from others, saw on a news show). Each device and program required teams of analysts, designers, developers, testers, trainers, and production engineers to bring them to fruition. Choose the top three that you would have found interesting to work on and consider which team role(s) you would have found most rewarding.

Exercise 13.4 Extend the CS timeline found in Appendix C on page 207 to add more CS milestones, particularly after 2011, that you find interesting or noteworthy.

Appendix A

Contributions

Contributor List for "Python for Informatics"

Bruce Shields for copy editing early drafts, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

Preface for "Think Python"

The strange history of "Think Python"

(Allen B. Downey)

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize the jargon and define each term at first use.
- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book.

Jeff and I revised the book, incorporated a case study by Chris Meyers, and in 2001 we released *How to Think Like a Computer Scientist: Learning with Python*, also under the GNU Free Documentation License. As Green Tea Press, I published the book and started selling hard copies through Amazon.com and college book stores. Other books from Green Tea Press are available at `greenteapress.com`.

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Over the last five years I have continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises. In 2008 I started work on a major revision—at the same time, I was contacted by an editor at Cambridge University Press who was interested in publishing the next edition. Good timing!

I hope you enjoy working with this book, and that it helps you learn to program and think, at least a little bit, like a computer scientist.

Acknowledgements for "Think Python"

(Allen B. Downey)

First and most importantly, I thank Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

I also thank Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

And I thank the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible.

I also thank the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

I thank all the students who worked with earlier versions of this book and all the contributors (listed in an Appendix) who sent in corrections and suggestions.

And I thank my wife, Lisa, for her work on this book, and Green Tea Press, and everything else, too.

Allen B. Downey
Needham MA

Allen Downey is an Associate Professor of Computer Science at the Franklin W. Olin College of Engineering.

Contributor List for "Think Python"

(Allen B. Downey)

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

For the detail on the nature of each of the contributions from these individuals, see the "Think Python" text.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardio, and Paul Stoop.

Appendix B

Copyright Detail

I would have preferred to license the book under the less restrictive CC-BY-SA license. But unfortunately there are a few unscrupulous organizations who search for and find freely licensed books, and then publish and sell virtually unchanged copies of the books on a print on demand service such as LuLu or CreateSpace. CreateSpace has (thankfully) added a policy that gives the wishes of the actual copyright holder preference over a non-copyright holder attempting to publish a freely licensed work. Unfortunately there are many print-on-demand services and very few have as well-considered a policy as CreateSpace.

Regretfully, I added the NC element to the license this book to give me recourse in case someone tries to clone this book and sell it commercially. Unfortunately, adding NC limits uses of this material that I would like to permit. So I have added this section of the document to describe specific situations where I am giving my permission in advance to use the material in this book in situations that some might consider commercial.

- If you are printing a limited number of copies of all or part of this book for use in a course (e.g., like a coursepack), then you are granted CC-BY license to these materials for that purpose.
- If you are a teacher at a university and you translate this book into a language other than English and teach using the translated book, then you can contact me and I will granted you a CC-BY-SA license to these materials with respect to the publication of your translation. In particular, you will be permitted to sell the resulting translated book commercially.

If you are intending to translate the book, you may want to contact me so we can make sure that you have all of the related course materials so you can translate them as well.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Appendix C

Timeline of Computer Science

The timeline on the following pages depicts some milestones from CS's long history. It is based on a curated list from Scott Aaronson[1]. With its intention to collapse thousands of years of events into a few pages, the timeline is interesting but necessarily incomplete. Note that the timeline ends in 2011. Exercise 13.4 on page 199 asks you to bring it up to date.

[1]Taken from Shtetl-Optimized: The Blog of Scott Aaronson; https://www.scottaaronson.com/blog/?p=524

Figure C.1: Beginning through 1799

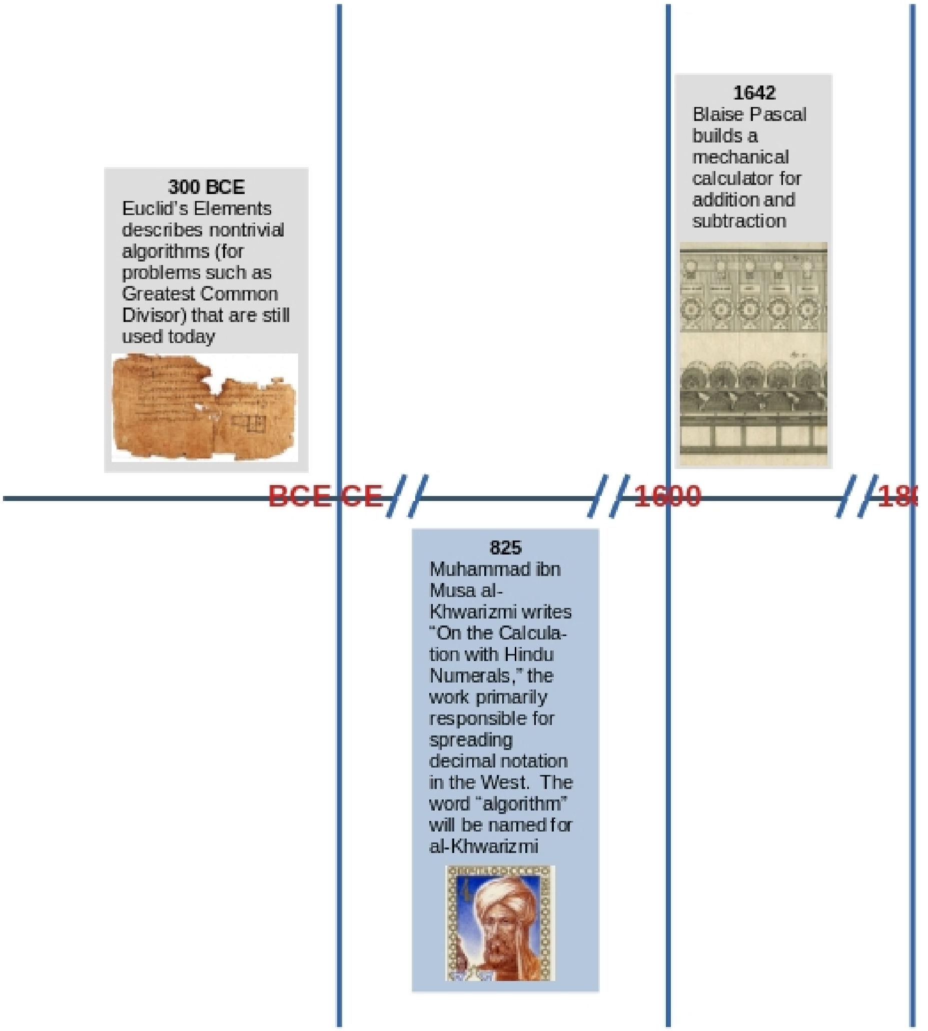


Figure C.2: 1800 through 1949

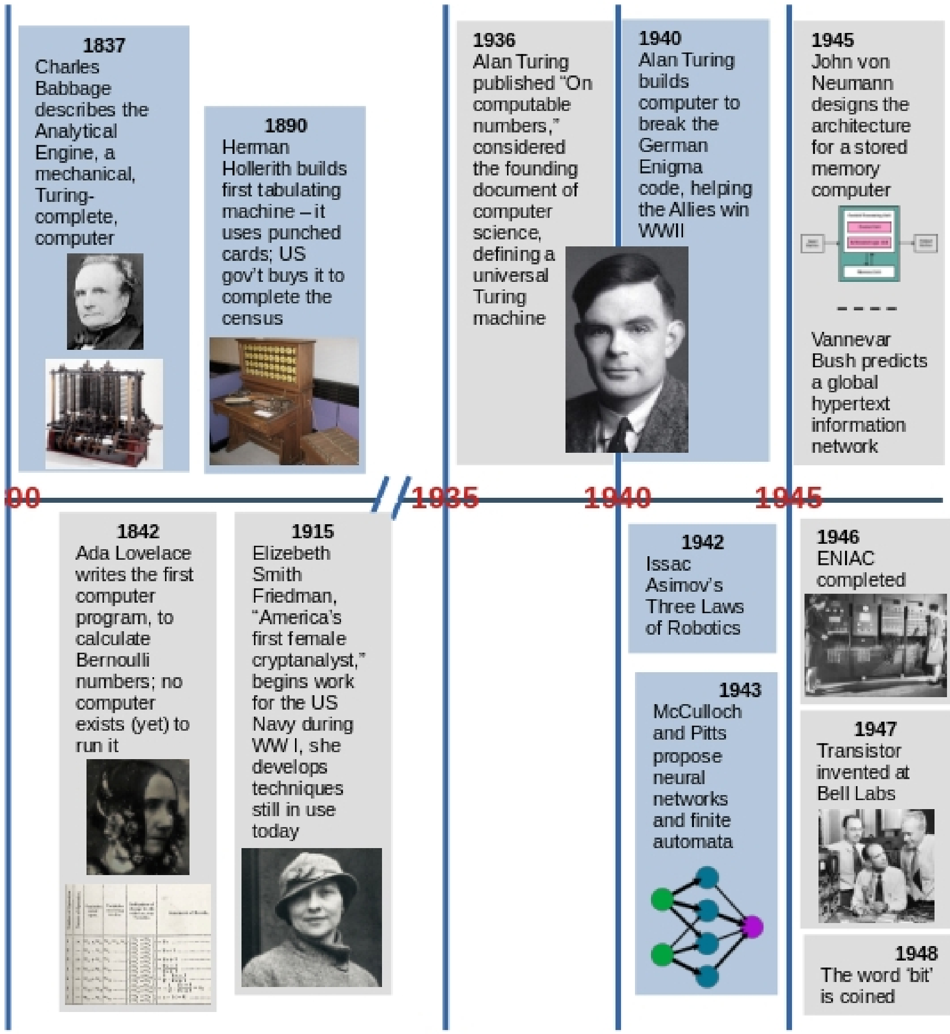


Figure C.3: 1950 through 1979

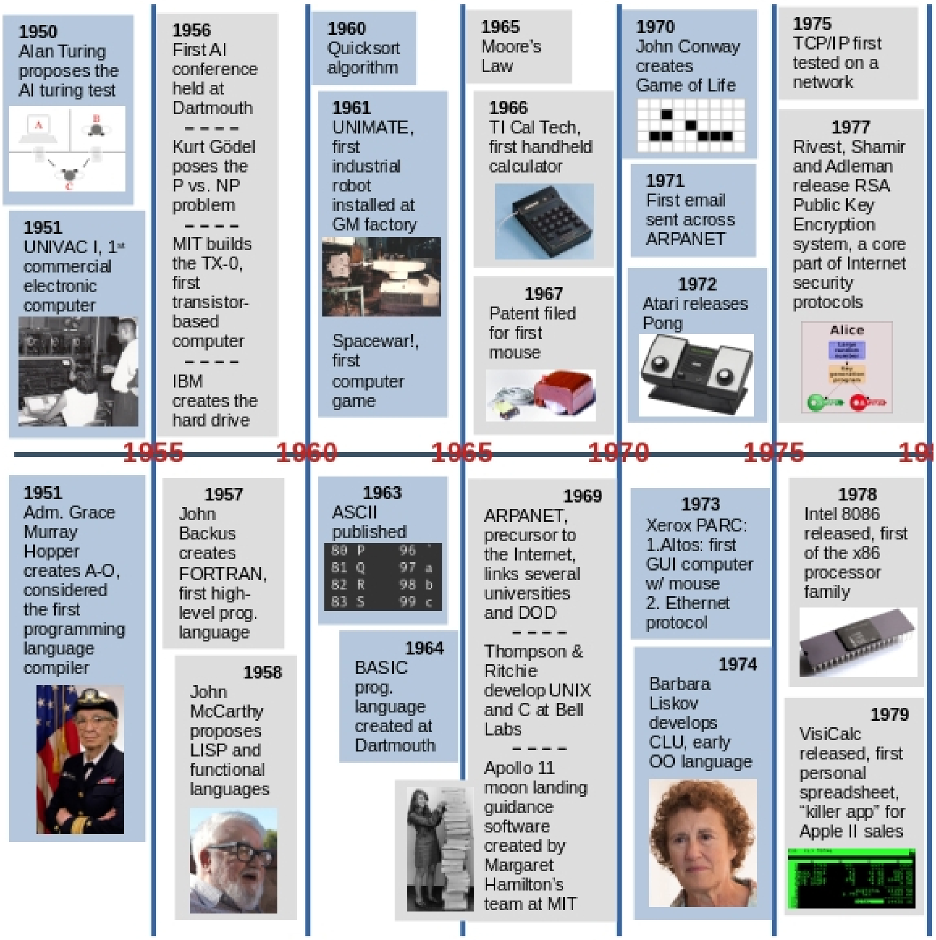
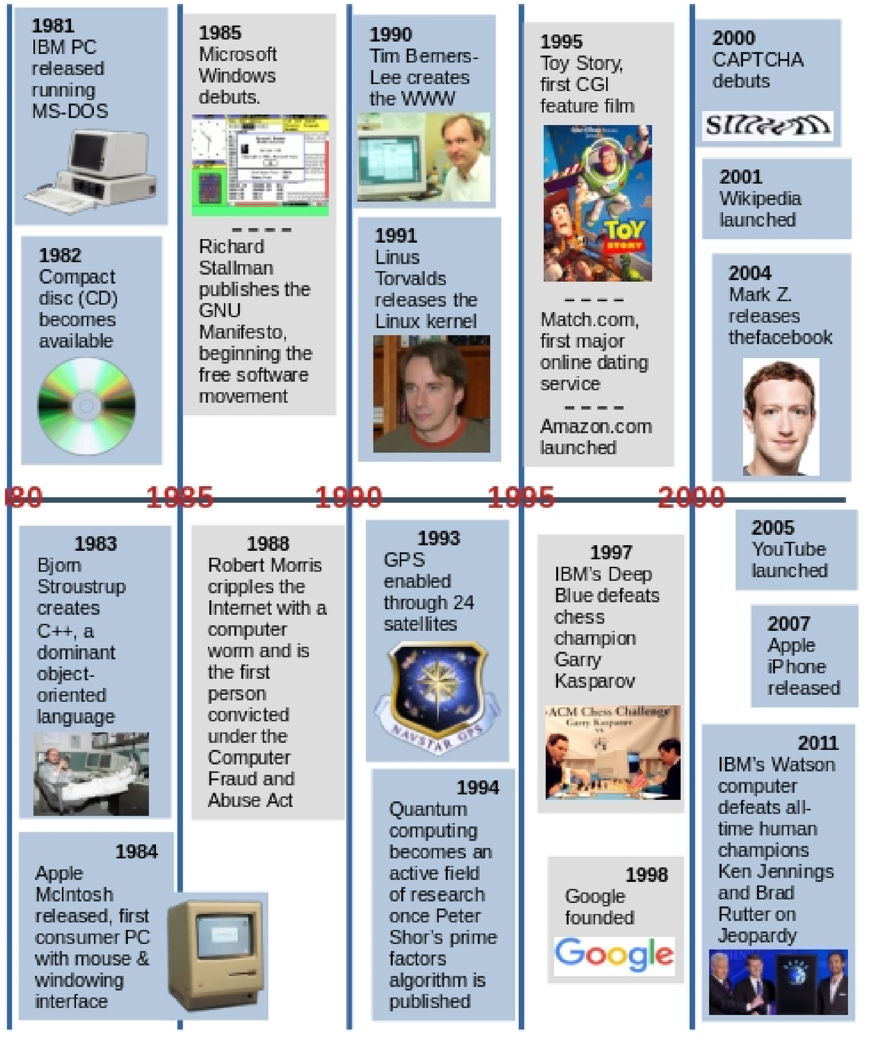


Figure C.4: 1980 through 2011



C.1 Image sources

Table C.1: Image Sources

Year	Item	Source
300	Euclid's Elements	https://en.wikipedia.org/wiki/Euclid%27s_Elements#/media/File:P._Oxy._I_29.jpg
825	Muhammad ibn Musa al-Khwarizmi	https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi#/media/File:1983_CPA_5426_(1).png
1642	Blaise Pascal Calculator	https://en.wikipedia.org/wiki/Pascal%27s_calculator#/media/File:Pascaline_-_top_view_and_mechanism.jpg
1837	Analytical Engine (portion)	https://en.wikipedia.org/wiki/Analytical_Engine#/media/File:Babbages_Analytical_Engine,_1834-1871._(9660574685).jpg
1837	Charles Babbage	https://en.wikipedia.org/wiki/Charles_Babbage#/media/File:Charles_Babbage_-_1860.jpg
1842	Ada Lovelace	https://en.wikipedia.org/wiki/Ada_Lovelace#/media/File:Ada_Byron_daguerreotype_by_Antoine_Claudet_1843_or_1850.jpg
1842	Ada Lovelace's Program	https://en.wikipedia.org/wiki/Ada_Lovelace#/media/File:Diagram_for_the_computation_of_Bernoulli_numbers.jpg
1890	Hollerith machine	https://en.wikipedia.org/wiki/File:HollerithMachine.CHM.jpg
1915	Elizebeth Smith Friedman	https://en.wikipedia.org/wiki/Elizebeth_Smith_Friedman#/media/File:Elizebeth-Friedman.jpg
1936	Alan Turing	https://www.newgon.net/wiki/Alan_Turing#/media/File:Turing.jpg
1943	Neural Network	https://en.wikipedia.org/wiki/Neural_network#/media/File:Neural_network_example.svg
1945	von Neumann Architecture	https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg
1946	ENIAC	https://en.wikipedia.org/wiki/ENIAC#/media/File:Two_women_operating_ENIAC_(full_resolution).jpg

Year	Item	Source
1947	Transistor	https://en.wikipedia.org/wiki/Transistor#/media/File:Bardeen_Shockley_Brattain_1948.JPG
1950	Turing test	https://en.wikipedia.org/wiki/Turing_test#/media/File:Turing_test_diagram.png
1951	Grace Hopper	https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg
1951	UNIVAC I	https://en.wikipedia.org/wiki/UNIVAC_I#/media/File:Univac_I_at_Census_Bureau_with_two_operators.jpg
1958	John McCarthy	https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)#/media/File:John_McCarthy_Stanford.jpg
1961	UNIMATE	https://www.nytimes.com/2011/08/16/business/george-devol-developer-of-robot-arm-dies-at-99.html
1963	ASCII	https://www.johndcook.com/blog/2022/05/28/how-to-memorize-the-ascii-table/
1966	TI Cal Tech	http://www.lokker.net/THOCP/hardware/ti_calculators.htm
1967	First mouse	https://en.wikipedia.org/wiki/Douglas_Engelbart#/media/File:SRI_Computer_Mouse.jpg
1969	Margaret Hamilton	https://en.wikipedia.org/wiki/Apollo_Guidance_Computer#/media/File:Margaret_Hamilton_-_restoration.jpg
1970	Conway's Game of Life	https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#/media/File:Game_of_life_acorn.svg
1972	Pong	https://en.wikipedia.org/wiki/Pong#/media/File:TeleGames-Atari-Pong.jpg
1974	Barbara Liskov	https://en.wikipedia.org/wiki/Barbara_Liskov#/media/File:Barbara_Liskov_MIT_computer_scientist_2010.jpg
1977	RSA/PKI	https://en.wikipedia.org/wiki/Public-key_cryptography#/media/File:Public-key-crypto-1.svg
1978	8086 processor	https://en.wikipedia.org/wiki/Intel_8086#/media/File:Intel_C8086.jpg

Year	Item	Source
1979	VisiCalc	https://en.wikipedia.org/wiki/VisiCalc#/media/File:Visicalc.png
1981	IBM PC	https://en.wikipedia.org/wiki/IBM_Personal_Computer#/media/File:IBM_PC-IMG_7271_(transparent).png
1982	CD	https://en.wikipedia.org/wiki/Compact_disc#/media/File:OD_Compact_disc.svg
1983	Bjorn Stroustrup	https://en.wikipedia.org/wiki/C%2B%2B#/media/File:BjarneStroustrup.jpg
1984	Macintosh	https://en.wikipedia.org/wiki/1984_(advertisement)#/media/File:Macintosh_128k_transparency.png
1985	MS Windows	https://en.wikipedia.org/wiki/Microsoft_Windows#/media/File:Windows1.0.png
1990	Tim Berners-Lee	https://home.cern/news/news/computing/web30-30-year-anniversary-invention-changed-world
1991	Linus Torvalds	https://www.linkedin.com/in/linustorvalds/overlay/photo/
1993	GPS	https://en.wikipedia.org/wiki/Global_Positioning_System
1995	Toy story	https://www.imdb.com/title/tt0114709/
1997	IBM Deep Blue	https://spectrum.ieee.org/how-ibms-deep-blue-beat-world-champion-chess-player-garry-kasparov
1998	Google	https://www.creativebloq.com/news/google-logo-history
2000	captcha	https://en.wikipedia.org/wiki/CAPTCHA#/media/File:Captcha.jpg
2004	Mark Z.	https://news.harvard.edu/gazette/story/2017/03/zuckerberg-speaks-at-harvards-366th-commencement/
2011	Watson on Jeopardy	https://www.csmonitor.com/Technology/Latest-News-Wires/2011/0215/On-Jeopardy-Watson-s-mistakes-reveal-its-genius

References

Berners-Lee, Tim (2014). "We need a Magna Carta for the Internet". en. In: *New Perspectives Quarterly* 31.3. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/npqu.11475, pp. 39–41. ISSN: 1540-5842. DOI: 10.1111/npqu.11475. URL: http://onlinelibrary.wiley.com/doi/abs/10.1111/npqu.11475 (visited on 06/07/2022).

Berners-Lee, Tim, Robert Cailliau, Jean-François Groff, and Bernd Pollermann (2010). "World-wide web: The information universe". English. In: *Internet Research* 20.4. Num Pages: 461-471 Place: Bradford, United Kingdom Publisher: Emerald Group Publishing Limited, pp. 461–471. ISSN: 10662243. DOI: https://doi.org/10.1108/10662241011059471. URL: https://www.proquest.com/docview/578103687/abstract/84A7534ED9F64921PQ/1 (visited on 06/07/2022).

Bi, Qifang, Katherine E Goodman, Joshua Kaminsky, and Justin Lessler (Dec. 2019). "What is machine learning? A primer for the epidemiologist". In: *American Journal of Epidemiology* 188.12, pp. 2222–2239. ISSN: 0002-9262. DOI: 10.1093/aje/kwz189. URL: https://doi.org/10.1093/aje/kwz189 (visited on 06/07/2022).

Chalaby, Jean K. (Feb. 2016). "Television and globalization: The TV content global value chain". In: *Journal of Communication* 66.1. Publisher: Oxford University Press / USA, pp. 35–59. ISSN: 00219916. DOI: 10.1111/jcom.12203. URL: http://ezproxy.baylor.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=ufh&AN=113138492&site=ehost-live&scope=site (visited on 06/07/2022).

Cirillo, Davide, Silvina Catuara-Solarz, Czuee Morey, Emre Guney, Laia Subirats, Simona Mellino, Annalisa Gigante, Alfonso Valencia, María José Rementeria, Antonella Santuccione Chadha, and Nikolaos Mavridis (June 2020). "Sex and gender differences and biases in artificial intelligence for biomedicine and healthcare". en. In: *NPJ Digital Medicine* 3.1. Number: 1 Publisher: Nature Publishing Group, pp. 1–11. ISSN: 2398-6352. DOI: 10.1038/s41746-020-0288-5. URL: https://www.nature.com/articles/s41746-020-0288-5 (visited on 05/24/2021).

Computer science: Definition and meaning (n.d.). URL: https://www.merriam-webster.com/dictionary/computer%20science (visited on 01/08/2023).

De-Sola Gutiérrez, José, Fernando Rodríguez de Fonseca, and Gabriel Rubio (2016). "Cell-phone addiction: A review". In: *Frontiers in Psychiatry* 7. ISSN: 1664-0640. URL: https://www.frontiersin.org/article/10.3389/fpsyt.2016.00175 (visited on 06/07/2022).

Dienlin, Tobias and Sabine Trepte (Apr. 2015). "Is the privacy paradox a relic of the past? An in-depth analysis of privacy attitudes and privacy behaviors". In: *European Journal of Social Psychology* 45.3. Publisher: John Wiley & Sons, Inc., pp. 285–297. ISSN: 00462772. DOI: 10.1002/ejsp.2049. URL: http://ezproxy.baylor.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=pbh&AN=102271752&site=ehost-live&scope=site (visited on 06/07/2022).

Fennelly, Orna (Mar. 2021). *For AI to be effective in healthcare, gender bias must be addressed.* en. URL: https://www.siliconrepublic.com/innovation/ai-healthcare-gender-bias (visited on 05/24/2021).

Koya, Kushwanth and Gobinda Chowdhury (Jan. 2020). "Measuring impact of academic research in computer and information science on society". In: *Proceedings of the 2020 2nd Asia Pacific information technology conference*. APIT 2020. New York, NY, USA: Association for Computing Machinery, pp. 78–85. ISBN: 978-1-4503-7685-3. URL: http://doi.org/10.1145/3379310.3379312 (visited on 06/07/2022).

Ma, Liye and Baohong Sun (Sept. 2020). "Machine learning and AI in marketing – Connecting computing power to human insights". en. In: *International Journal of Research in Marketing* 37.3, pp. 481–504. ISSN: 0167-8116. DOI: 10.1016/j.ijresmar.2020.04.005. URL: https://www.sciencedirect.com/science/article/pii/S0167811620300410 (visited on 06/07/2022).

McNeff, J.G. (Mar. 2002). "The global positioning system". In: *IEEE Transactions on Microwave Theory and Techniques* 50.3. Conference Name: IEEE Transactions on Microwave Theory and Techniques, pp. 645–652. ISSN: 1557-9670. DOI: 10.1109/22.989949.

Read, David S. (2023). "Why women go elsewhere: A phenomenological study of women's underrepresentation in computer science". English. Ed.D. United States – Texas: Baylor University.

Resource description framework (RDF): Concepts and abstract syntax (n.d.). URL: https://www.w3.org/TR/rdf-concepts/ (visited on 06/07/2022).

Vinichenko, Mikhail V., Galina Yu. Nikiporets-Takigawa, Oksana L. Chulanova, and Natalia V. Ljapunova (Oct. 2021). "Threats and risks from the digitalization of society and artificial intelligence: Views of generation Z students".

en. In: *International Journal of Advanced and Applied Sciences* 8.10. Publisher: IASE, pp. 108–115. ISSN: 2313-626X. DOI: 10.21833/ijaas.2021.10.012. URL: http://www.science-gate.com/IJAAS/2021/V8I10/1021833ijaas202110012.html (visited on 01/04/2022).

Wiens, Jenna, I. I. W. Nicholson Price, and Michael W. Sjoding (Jan. 2020). "Diagnosing bias in data-driven algorithms for healthcare." English. In: *Nature Medicine* 26.1. Publisher: Nature Publishing Group, pp. 25–27. ISSN: 10788956. DOI: 10.1038/s41591-019-0726-6. URL: http://go.gale.com/ps/i.do?p=AONE&sw=w&issn=10788956&v=2.1&it=r&id=GALE%7CA611371145&sid=googleScholar&linkaccess=abs (visited on 01/04/2022).

Zhou, Xinyi and Reza Zafarani (Sept. 2020). "A survey of fake news: Fundamental theories, detection methods, and opportunities". In: *ACM Computing Surveys* 53.5, 109:1–109:40. ISSN: 0360-0300. DOI: 10.1145/3395046. URL: http://doi.org/10.1145/3395046 (visited on 01/04/2022).

Index