Chapter 1. A Taste of Py

Only ugly languages become popular. Python is the one exception.

—Don Knuth

Mysteries

Let's begin with two mini-mysteries and their solutions. What do you think the following two lines mean?

```
(Row 1): (RS) K18,ssk,k1,turn work.
(Row 2): (WS) Sl 1 pwise,p5,p2tog,p1,turn.
```

It looks technical, like some kind of computer program. Actually, it's a *knitting pattern*; specifically, a fragment describing how to turn the heel of a sock, like the one in Figure 1-1.

Figure 1-1. Knitted socks

This makes as much sense to me as a Sudoku puzzle does to one of my cats, but my wife understands it perfectly. If you're a knitter, you do, too.

Let's try another mysterious text, found on an index card. You'll figure out

its purpose right away, although you might not know its final product:

```
  1/2 c. butter or margarine
  1/2 c. cream
  2 1/2 c. flour
  1 t. salt
  1 T. sugar
  4 c. riced potatoes (cold)

Be sure all ingredients are cold before adding flour.
Mix all ingredients.
Knead thoroughly.
Form into 20 balls.  Store cold until the next step.
For each ball:
  Spread flour on cloth.
  Roll ball into a circle with a grooved rolling pin.
  Fry on griddle until brown spots appear.
  Turn over and fry other side.
```

Even if you don't cook, you probably recognized that it's a *recipe*[1]: a list of food ingredients followed by directions for preparation. But what does it make? It's *lefse*, a Norwegian delicacy that resembles a tortilla (Figure 1-2). Slather on some butter and jam or whatever you like, roll it up, and enjoy.

Figure 1-2. Lefse

The knitting pattern and the recipe share some features:

- A regular *vocabulary* of words, abbreviations, and symbols. Some might be familiar, others mystifying.
- Rules about what can be said, and where—*syntax*.
- A *sequence of operations* to be performed in order.
- Sometimes, a repetition of some operations (a *loop*), such as the method for frying each piece of lefse.
- Sometimes, a reference to another sequence of operations (in computer terms, a *function*). In the recipe, you might need to refer to another recipe for ricing potatoes.
- Assumed knowledge about the *context*. The recipe assumes you that know what water is and how to boil it. The knitting pattern assumes that you can knit and purl without stabbing yourself too often.
- Some *data* to be used, created, or modified—potatoes and yarn.
- The *tools* used to work with the data—pots, mixers, ovens, knitting sticks.
- An expected *result*. In our examples, something for your feet and something for your stomach. Just don't mix them up.

Whatever you call them—idioms, jargon, little languages—you see examples of them everywhere. The lingo saves time for people who know it,

while mystifying the rest of us. Try deciphering a newspaper column about bridge if you don't play the game, or a scientific paper if you're not a scientist (or even if you are, but in a different field).

Little Programs

You'll see all of these ideas in computer programs, which are themselves like little languages, specialized for humans to tell computers what to do. I used the knitting pattern and recipe to demonstrate that programming isn't that mysterious. It's largely a matter of learning the right words and the rules.

Now, it helps greatly if there aren't too many words and rules, and if you don't need to learn too many of them at once. Our brains can hold only so much at one time.

Let's finally see a real computer program (Example 1-1). What do you think this does?

Example 1-1. countdown.py

```
for countdown in 5, 4, 3, 2, 1, "hey!":
    print(countdown)
```

If you guessed that it's a Python program that prints the lines

```
5
4
3
2
1
hey!
```

then you know that Python can be easier to learn than a recipe or knitting pattern. And you can practice writing Python programs from the comfort and safety of your desk, far from the hazards of hot water and pointy sticks.

The Python program has some special words and symbols— `for` , `in` ,

`print`, commas, colons, parentheses, and so on—that are important parts of the language's *syntax* (rules). The good news is that Python has a nicer syntax, and less of it to remember, than most computer languages. It seems more natural—almost like a recipe.

Example 1-2 is another tiny Python program; it selects one Harry Potter spell from a Python *list* and prints it.

Example 1-2. spells.py

```
spells = [
    "Riddikulus!",
    "Wingardium Leviosa!",
    "Avada Kedavra!",
    "Expecto Patronum!",
    "Nox!",
    "Lumos!",
    ]
print(spells[3])
```

The individual spells are Python *strings* (sequences of text characters, enclosed in quotes). They're separated by commas and enclosed in a Python *list* that's defined by enclosing square brackets (`[` and `]`). The word `spells` is a *variable* that gives the list a name so that we can do things with it. In this case, the program would print the fourth spell:

```
Expecto Patronum!
```

Why did we say `3` if we wanted the fourth? A Python list such as `spells` is a sequence of values, accessed by their *offset* from the beginning of the list. The first value is at offset `0`, and the fourth value is at offset `3`.

NOTE

People count from 1, so it might seem weird to count from 0. It helps to think in terms of offsets instead of positions. Yes, this is an example of how computer programs sometimes differ from common language usage.

Lists are very common *data structures* in Python, and Chapter 7 shows

how to use them.

The program in Example 1-3 prints a quote from one of the Three Stooges, but referenced by who said it rather than its position in a list.

Example 1-3. quotes.py

```
quotes = {
    "Moe": "A wise guy, huh?",
    "Larry": "Ow!",
    "Curly": "Nyuk nyuk!",
    }
stooge = "Curly"
print(stooge, "says:", quotes[stooge])
```

If you were to run this little program, it would print the following:

```
Curly says: Nyuk nyuk!
```

`quotes` is a variable that names a Python *dictionary*—a collection of unique *keys* (in this example, the name of the Stooge) and associated *values* (here, a notable saying of that Stooge). Using a dictionary, you can store and look up things by name, which is often a useful alternative to a list.

The `spells` example used square brackets (`[` and `]`) to make a Python list, and the `quotes` example uses curly brackets (`{` and `}`, which are no relation to Curly), to make a Python dictionary. Also, a colon (`:`) is used to associate each key in the dictionary with its value. You can read much more about dictionaries in Chapter 8.

That wasn't too much syntax at once, I hope. In the next few chapters, you'll encounter more of these little rules, a bit at a time.

A Bigger Program

And now for something completely different: Example 1-4 presents a Python program performing a more complex series of tasks. Don't expect to understand how the program works yet; that's what this book is for!

The intent is to introduce you to the look and feel of a typical nontrivial Python program. If you know other computer languages, evaluate how Python compares. Even without knowing Python yet, can you roughly figure out what each line does before reading the explanation after the program? You've already seen examples of a Python list and a dictionary, and this throws in a few more features.

In earlier printings of this book, the sample program connected to a YouTube website and retrieved information on its most highly rated videos, like "Charlie Bit My Finger." It worked well until shortly after the ink was dry on the second printing. That's when Google dropped support for this service and the marquee sample program stopped working. Our new Example 1-4 goes to another site which should be around longer—the *Wayback Machine* at the Internet Archive, a free service that has saved billions of web pages (and movies, TV shows, music, games, and other digital artifacts) over 20 years. You'll see more examples of such *web APIs* in Chapter 18.

The program will ask you to type a URL and a date. Then, it asks the Wayback Machine if it has a copy of that website around that date. If it found one, it returns the information to this Python program, which prints the URL and displays it in your web browser. The point is to show how Python handles a variety of tasks—get your typed input, communicate across the internet to a website, get back some content, extract a URL from it, and convince your web browser to display that URL.

If we got back a normal web page full of HTML-formatted text, we would need to figure out how to display it, which is a lot of work that we happily entrust to web browsers. We could also try to extract the parts that we want (see more details about *web scraping* in Chapter 18). Either choice would be more work and a larger program. Instead, the Wayback Machine returns data in *JSON* format. JSON (JavaScript Object Notation) is a human-readable text format that describes the types, values, and order of the data within it. It's another little language, and it has become a popular way to exchange data among different computer languages and systems. You'll read more about JSON in Chapter 12.

Python programs can translate JSON text into Python *data structures*—the kind you'll see in the next few chapters—as though you wrote a program to create them yourself. Our little program just selects one piece (the URL

of the old page from the Internet Archive website). Again, this is a complete Python program that you can run yourself. We've included only a little error-checking, just to keep the example short. The line numbers are not part of the program; they are included to help you follow the description that we provide after the program.

Example 1-4. archive.py

```
1 import webbrowser
2 import json
3 from urllib.request import urlopen
4
5 print("Let's find an old website.")
6 site = input("Type a website URL: ")
7 era = input("Type a year, month, and day, like 20150613: ")
8 url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
9 response = urlopen(url)
10 contents = response.read()
11 text = contents.decode("utf-8")
12 data = json.loads(text)
13 try:
14     old_site = data["archived_snapshots"]["closest"]["url"]
15     print("Found this copy: ", old_site)
16     print("It should appear in your browser now.")
17     webbrowser.open(old_site)
18 except:
19     print("Sorry, no luck finding", site)
```

This little Python program did a lot in a few fairly readable lines. You don't know all these terms yet, but you will within the next few chapters. Here's what's going on in each line:

1. *Import* (make available to this program) all the code from the Python *standard library* module called `webbrowser`.
2. Import all the code from the Python standard library module called `json`.
3. Import only the `urlopen` *function* from the standard library module `urllib.request`.
4. A blank line, because we don't want to feel crowded.
5. Print some initial text to your display.
6. Print a question about a URL, read what you type, and save it in a program *variable* called `site`.

7. Print another question, this time reading a year, month, and day, and then save it in a variable called `era`.
8. Construct a string variable called `url` to make the Wayback Machine look up its copy of the site and date that you typed.
9. Connect to the web server at that URL and request a particular *web service*.
10. Get the response data and assign to the variable `contents`.
11. *Decode* `contents` to a text string in JSON format, and assign to the variable `text`.
12. Convert `text` to `data`—Python data structures.
13. Error-checking: `try` to run the next four lines, and if any fail, run the last line of the program (after the `except`).
14. If we got back a match for this site and date, extract its value from a three-level Python *dictionary*. Notice that this line and the next two are indented. That's how Python knows that they go with the preceding `try` line.
15. Print the URL that we found.
16. Print what will happen after the next line executes.
17. Display the URL we found in your web browser.
18. If anything failed in the previous four lines, Python jumps down to here.
19. If it failed, print a message and the site that we were looking for. This is indented because it should be run only if the preceding `except` line runs.

When I ran this in a terminal window, I typed a site URL and a date, and got this text output:

```
$ python archive.py
Let's find an old website.
Type a website URL: lolcats.com
Type a year, month, and day, like 20150613: 20151022
Found this copy: http://web.archive.org/web/20151102055938/http://www.lolcats.co
It should appear in your browser now.
```

And Figure 1-3 shows what appeared in my browser.

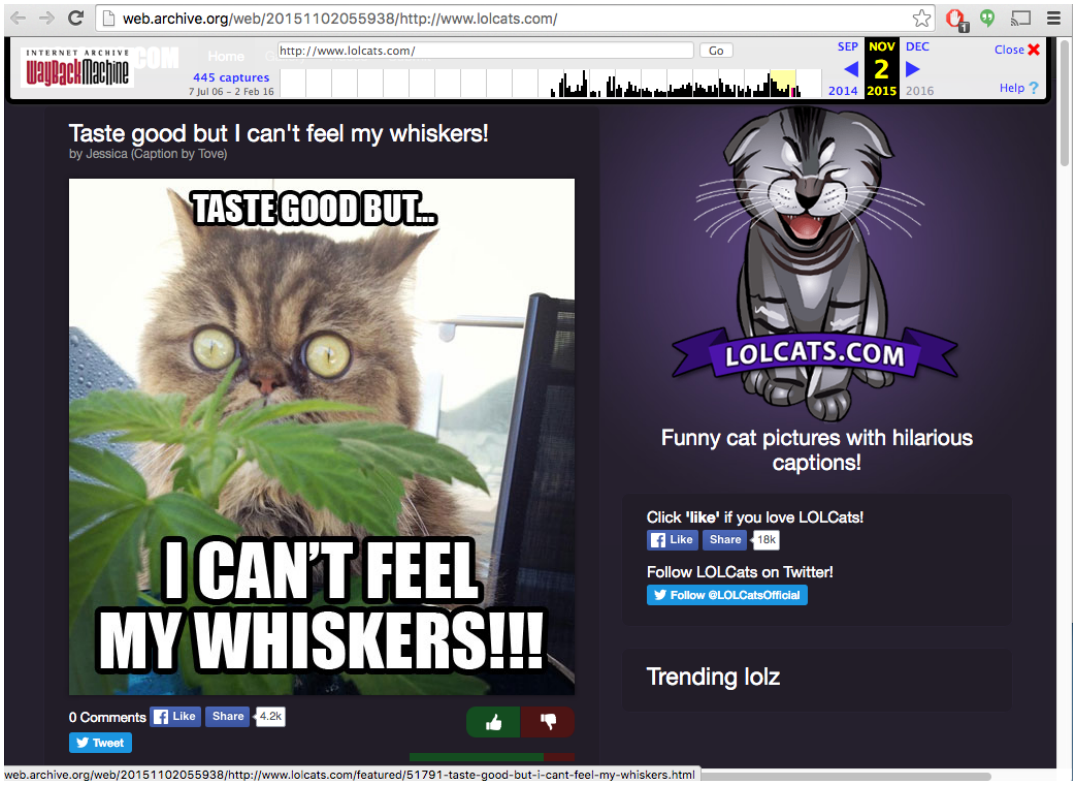

Figure 1-3. From the Wayback Machine

In the previous example, we used some of Python's *standard library* modules (programs that are included with Python when it's installed), but there's nothing sacred about them. Python has a trove of excellent third-party software. Example 1-5 is a rewrite that accesses the Internet Archive website with an external Python software package called `requests`.

Example 1-5. archive2.py

```
import webbrowser
import requests

print("Let's find an old website.")
site = input("Type a website URL: ")
era = input("Type a year, month, and day, like 20150613: ")
url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
response = requests.get(url)
data = response.json()
try:
    old_site = data["archived_snapshots"]["closest"]["url"]
    print("Found this copy: ", old_site)
    print("It should appear in your browser now.")
    webbrowser.open(old_site)
except:
```

```
16     print("Sorry, no luck finding", site)
```

The new version is shorter, and I'd guess it's more readable for most people. You'll read more about `requests` in Chapter 18, and externally authored Python software in general in Chapter 11.

Python in the Real World

So, is learning Python worth the time and effort? Python has been around since 1991 (older than Java, younger than C), and is consistently in the top five most popular computing languages. People are paid to write Python programs—serious stuff that you use every day, such as Google, YouTube, Instagram, Netflix, and Hulu. I've used it for production applications in many areas. Python has a reputation for productivity that appeals to fast-moving organizations.

You'll find Python in many computing environments, including these:

- The command line in a monitor or terminal window
- Graphical user interfaces (GUIs), including the web
- The web, on the client and server sides
- Backend servers supporting large popular sites
- The *cloud* (servers managed by third parties)
- Mobile devices
- Embedded devices

Python programs range from one-off *scripts*—such as those you've seen so far in this chapter—to million-line systems.

The 2018 Python Developers' Survey has numbers and graphs on Python's current place in the computing world.

We'll look at its uses in websites, system administration, and data manipulation. In the final chapters, we'll see specific uses of Python in the arts, science, and business.

Python Versus the Language from

Planet X

How does Python compare against other languages? Where and when would you choose one over the other? In this section, I show code samples from other languages, just so you can see what the competition looks like. You are *not* expected to understand these if you haven't worked with them. (By the time you get to the final Python sample, you might be relieved that you haven't had to work with some of the others.)

Each program is supposed to print a number and say a little about the language.

If you use a terminal or terminal window, the program that reads what you type, runs it, and displays the results is called the *shell* program. The Windows shell is called `cmd`; it runs *batch* files with the suffix `.bat`. Linux and other Unix-like systems (including macOS) have many shell programs. The most popular is called `bash` or `sh`. The shell has simple abilities, such as simple logic and expanding wildcard symbols such as `*` into filenames. You can save commands in files called *shell scripts* and run them later. These might be the first programs you encountered as a programmer. The problem is that shell scripts don't scale well beyond a few hundred lines, and they are much slower than the alternative languages. The next snippet shows a little shell program:

```
#!/bin/sh
language=0
echo "Language $language: I am the shell. So there."
```

If you saved this in a file as `test.sh` and ran it with `sh test.sh`, you would see the following on your display:

```
Language 0: I am the shell. So there.
```

Old stalwarts C and C++ are fairly low-level languages, used when speed is most important. Your operating system and many of its programs (including the `python` program on your computer) are probably written in C or C++.

These two are harder to learn and maintain. You need to keep track of many details like *memory management*, which can lead to program crashes and problems that are difficult to diagnose. Here's a little C program:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int language = 1;
    printf("Language %d: I am C! See? Si!\n", language);
    return 0;
}
```

C++ has the C family resemblance but has evolved some distinctive features:

```
#include <iostream>
using namespace std;
int main() {
    int language = 2;
    cout << "Language " << language << \
       ": I am C++!  Pay no attention to my little brother!" << \
       endl;
    return(0);
}
```

Java and C# are successors to C and C++ that avoid some of their forebears' problems—especially memory management—but can be somewhat verbose. The example that follows shows some Java:

```
public class Anecdote {
    public static void main (String[] args) {
        int language = 3;
        System.out.format("Language %d: I am Java! So there!\n", language);
    }
}
```

If you haven't written programs in any of these languages, you might wonder: what is all that *stuff*? We only wanted to print a simple line. Some languages carry substantial syntactic baggage. You'll learn more about this in Chapter 2.

C, C++, and Java are examples of *static languages*. They require you to specify some low-level details like data types for the computer. Appendix A shows how a data type like an integer has a specific number of bits in your computer, and can only do integer-ey things. In contrast, *dynamic languages* (also called *scripting languages*) do not force you to declare variable types before using them.

The all-purpose dynamic language for many years was Perl. Perl is very powerful and has extensive libraries. Yet, its syntax can be awkward, and the language seems to have lost momentum in the past few years to Python and Ruby. This example regales you with a Perl bon mot:

```
my $language = 4;
print "Language $language: I am Perl, the camel of languages.\n";
```

Ruby is a more recent language. It borrows a little from Perl, and is popular mostly because of *Ruby on Rails*, a web development framework. It's used in many of the same areas as Python, and the choice of one or the other might boil down to a matter of taste, or available libraries for your particular application. Here's a Ruby snippet:

```
language = 5
puts "Language #{language}: I am Ruby, ready and aglow."
```

PHP, which you can see in the example that follows, is very popular for web development because it makes it easy to combine HTML and code. However, the PHP language itself has a number of gotchas, and PHP has not caught on as a general language outside of the web. Here's what it looks like:

```
<?PHP
$language = 6;
echo "Language $language: I am PHP, a language and palindrome.\n";
?>
```

Go (or *Golang*, if you're trying to Google it) is a recent language that tries to be both efficient and friendly:

```
package main

import "fmt"

func main() {
  language := 7
  fmt.Printf("Language %d: Hey, ho, let's Go!\n", language)
}
```

Another modern alternative to C and C++ is Rust:

```
fn main() {
    println!("Language {}: Rust here!", 8)
```

Who's left? Oh yes, Python:

```
language = 9
print(f"Language {language}: I am Python. What's for supper?")
```

Why Python?

One reason, not necessarily the most important, is popularity. By various measures, Python is:

- The fastest-growing major programming language, as you can see in Figure 1-4.
- The editors of the June 2019 TIOBE Index say: "This month Python has reached again an all time high in TIOBE index of 8.5%. If Python can keep this pace, it will probably replace C and Java in 3 to 4 years time, thus becoming the most popular programming language of the world."
- Programming language of the year for 2018 (TIOBE), and top ranking by IEEE Spectrum and PyPL.
- The most popular language for introductory computer science courses at the top American colleges.
- The official teaching language for high schools in France.

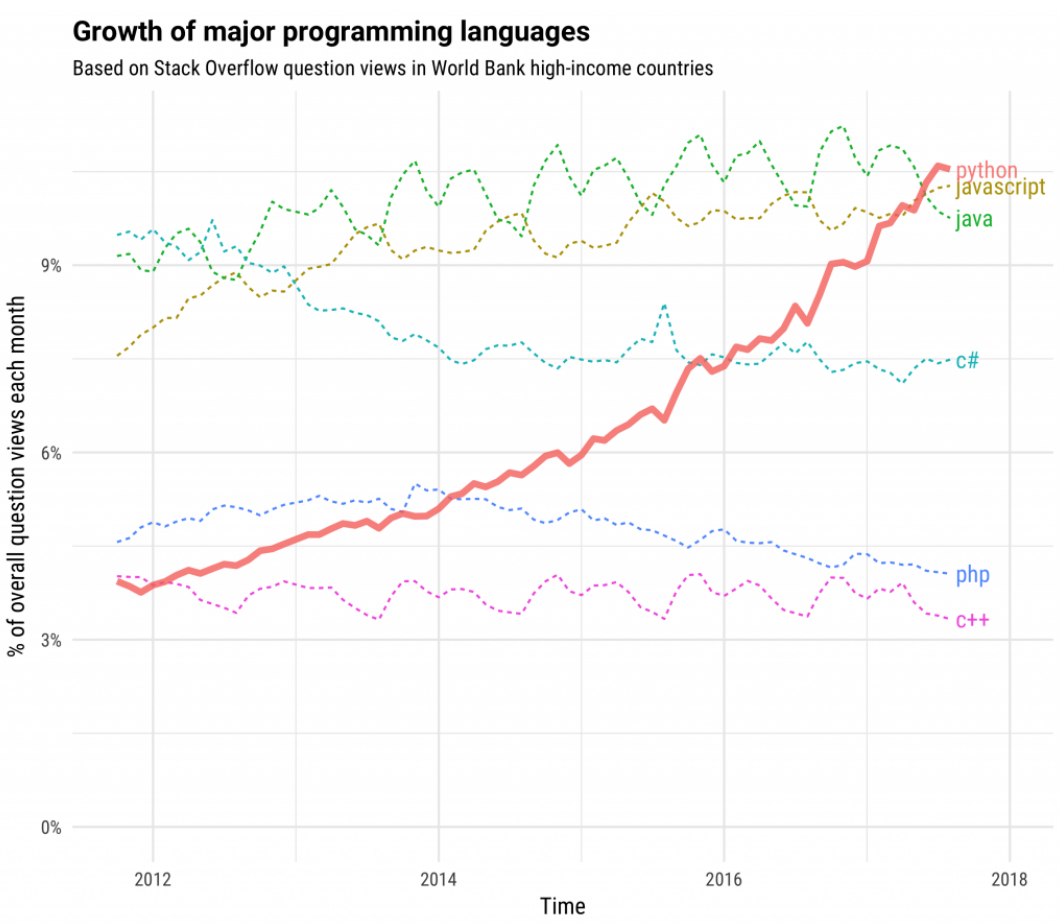

Figure 1-4. Python leads in major programming language growth

More recently, it's become extremely popular in the data science and machine learning worlds. If you want to land a well-paying programming job in an interesting area, Python is a good choice now. And if you're hiring, there's a growing pool of experienced Python developers.

But *why* is it popular? Programming languages don't exactly exude charisma. What are some underlying reasons?

Python is a good general-purpose, high-level language. Its design makes it very *readable*, which is more important than it sounds. Every computer program is written only once, but read and revised many times, often by many people. Being readable also makes it easier to learn and remember; hence, more *writable*. Compared with other popular languages, Python has a gentle learning curve that makes you productive sooner, yet it has depths that you can explore as you gain expertise.

Python's relative terseness makes it possible for you to write programs that are smaller than their equivalents in a static language. Studies have shown that programmers tend to produce roughly the same number of lines of code per day—regardless of the language—so, halving the lines of

code doubles your productivity, just like that. Python is the not-so-secret weapon of many companies that think this is important.

And of course, Python is free, as in beer (price) and speech (liberty). Write anything you want with Python, and use it anywhere, freely. No one can read your Python program and say, “That’s a nice little program you have there. It would be a shame if something happened to it.”

Python runs almost everywhere and has “batteries included”—a metric boatload of useful software in its standard library. This book presents many examples of the standard library and useful third-party Python code.

But, maybe the best reason to use Python is an unexpected one: people generally *enjoy* programming with it rather than seeing it as a necessary evil to get stuff done. It doesn’t get in the way. A familiar quote is that it “fits your brain.” Often, developers will say that they miss some Python design when they need to work in another language. And that separates Python from most of its peers.

Why Not Python?

Python isn’t the best language for every situation.

It is not installed everywhere by default. Appendix B shows you how to install Python if you don’t already have it on your computer.

It’s fast enough for most applications, but it might not be fast enough for some of the more demanding ones. If your program spends most of its time calculating things (the technical term is *CPU-bound*), a program written in C, C++, C#, Java, Rust, or Go will generally run faster than its Python equivalent. But not always!

Here are some solutions:

- Sometimes a better *algorithm* (a stepwise solution) in Python beats an inefficient one in C. The greater speed of development in Python gives you more time to experiment with alternatives.
- In many applications (notably, the web), a program twiddles its gos-

samer thumbs while awaiting a response from some server across a network. The CPU (central processing unit, the computer's *chip* that does all the calculating) is barely involved; consequently, end-to-end times between static and dynamic programs will be close.

- The standard Python interpreter is written in C and can be extended with C code. I discuss this a little in Chapter 19.
- Python interpreters are becoming faster. Java was terribly slow in its infancy, and a lot of research and money went into speeding it up. Python is not owned by a corporation, so its enhancements have been more gradual. In "PyPy", I talk about the *PyPy* project and its implications.
- You might have an extremely demanding application, and no matter what you do, Python doesn't meet your needs. The usual alternatives are C, C++, and Java. Go (which feels like Python but performs like C) or Rust could also be worth a look.

Python 2 Versus Python 3

One medium-sized complication is that there are two versions of Python out there. Python 2 has been around forever and is preinstalled on Linux and Apple computers. It has been an excellent language, but nothing's perfect. In computer languages, as in many other areas, some mistakes are cosmetic and easy to fix, whereas others are hard. Hard fixes are *incompatible*: new programs written with them will not work on the old Python system, and old programs written before the fix will not work on the new system.

Python's creator (Guido van Rossum) and others decided to bundle the hard fixes together, and introduced them as Python 3 in 2008. Python 2 is the past, and Python 3 is the future. The final version of Python 2 is 2.7, and it will be around for while, but it's the end of the line; there will be no Python 2.8. The end of Python 2 language support is in January of 2020. Security and other fixes will no longer be made, and many prominent Python packages will drop support for Python 2 by then. Operating systems will soon either drop Python 2 or make 3 their new default. Conversion of popular Python software to Python 3 had been gradual, but we're now well past the tipping point. All new development will be in Python 3.

This book is about Python 3. It looks almost identical to Python 2. The most obvious change is that `print` is a function in Python 3, so you need to call it with parentheses surrounding its arguments. The most important change is the handling of *Unicode* characters, which is covered in Chapter 12. I point out other significant differences as they come up.

Installing Python

Rather than cluttering this chapter, you can find the details on how to install Python 3 in Appendix B. If you don't have Python 3, or aren't sure, go there and see what you need to do for your computer. Yes, this is a pain in the wazoo (specifically, the right-anterior wazoo), but you'll need to do it only once.

Running Python

After you have installed a working copy of Python 3, you can use it to run the Python programs in this book as well as your own Python code. How do you actually run a Python program? There are two main ways:

- Python's built-in *interactive interpreter* (also called its *shell*) is the easy way to experiment with small programs. You type commands line by line and see the results immediately. With the tight coupling between typing and seeing, you can experiment faster. I'll use the interactive interpreter to demonstrate language features, and you can type the same commands in your own Python environment.
- For everything else, store your Python programs in text files, normally with the *.py* extension, and run them by typing `python` followed by those filenames.

Let's try both methods now.

Using the Interactive Interpreter

Most of the code examples in this book use the built-in interactive interpreter. When you type the same commands as you see in the examples and get the same results, you'll know you're on the right track.

You start the interpreter by typing just the name of the main Python program on your computer: it should be `python`, `python3`, or something similar. For the rest of this book, we assume it's called `python`; if yours has a different name, type that wherever you see `python` in a code example.

The interactive interpreter works almost exactly the same as Python works on files, with one exception: when you type something that has a value, the interactive interpreter prints its value for you automatically. This isn't a part of the Python language, just a feature of the interpreter to save you from typing `print()` all the time. For example, if you start Python and type the number `27` in the interpreter, it will be echoed to your terminal (if you have the line `27` in a file, Python won't get upset, but you won't see anything print when you run the program):

```
$ python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 27
27
```

NOTE

In the preceding example, `$` is a sample system *prompt* for you to type a command like `python` in the terminal window. We use it for the code examples in this book, although your prompt might be different.

By the way, `print()` also works within the interpreter whenever you want to print something:

```
>>> print(27)
27
```

If you tried these examples with the interactive interpreter and saw the same results, you just ran some real (though tiny) Python code. In the next few chapters, you'll graduate from one-liners to longer Python programs.

Using Python Files

If you put `27` in a file by itself and run it through Python, it will run, but it won't print anything. In normal noninteractive Python programs, you need to call the `print` function to print things:

```
print(27)
```

Let's make a Python program file and run it:

1. Open your text editor.
2. Type the line `print(27)`, as it appears here.
3. Save this to a file called *test.py*. Make sure you save it as plain text rather than a "rich" format such as RTF or Word. You don't need to use the *.py* suffix for your Python program files, but it does help you remember what they are.
4. If you're using a GUI—that's almost everyone—open a terminal window.[2]
5. Run your program by typing the following:

```
$ python test.py
```

You should see a single line of output:

```
27
```

Did that work? If it did, congratulations on running your first standalone Python program.

What's Next?

You'll be typing commands to an actual Python system, and they need to follow legal Python syntax. Rather than dumping the syntax rules on you all at once, we stroll through them over the next few chapters.

The basic way to develop Python programs is by using a plain-text editor and a terminal window. I use plain-text displays in this book, sometimes showing interactive terminal sessions and sometimes pieces of Python

files. You should know that there are also many good *integrated development environments* (IDEs) for Python. These may feature GUIs with advanced text editing and help displays. You can learn about details for some of these in Chapter 19.

Your Moment of Zen

Each computing language has its own style. In the Preface, I mentioned that there is often a *Pythonic* way to express yourself. Embedded in Python is a bit of free verse that expresses the Python philosophy succinctly (as far as I know, Python is the only language to include such an Easter egg). Just type `import this` into your interactive interpreter and then press the Enter key whenever you need this moment of Zen:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea--let's do more of those!
```

I'll bring up examples of these sentiments throughout the book.

Coming Up

The next chapter talks about Python data types and variables. This will prepare you for the following chapters, which delve into Python's data types and code structures in detail.

Things to Do

This chapter was an introduction to the Python language—what it does, how it looks, and where it fits in the computing world. At the end of each chapter, I suggest some mini-projects to help you remember what you just read and prepare you for what's to come.

1.1 If you don't already have Python 3 installed on your computer, do it now. Read Appendix B for the details for your computer system.

1.2 Start the Python 3 interactive interpreter. Again, details are in Appendix B. It should print a few lines about itself and then a single line starting with `>>>`. That's your prompt to type Python commands.

1.3 Play with the interpreter a little. Use it like a calculator and type this: `8 * 9`. Press the Enter key to see the result. Python should print `72`.

1.4 Type the number `47` and press the Enter key. Did it print `47` for you on the next line?

1.5 Now, type `print(47)` and press Enter. Did that also print `47` for you on the next line?

[1] Usually only found in cookbooks and cozy mysteries.

[2] If you're not sure what this means, see Appendix B for details for different operating systems.