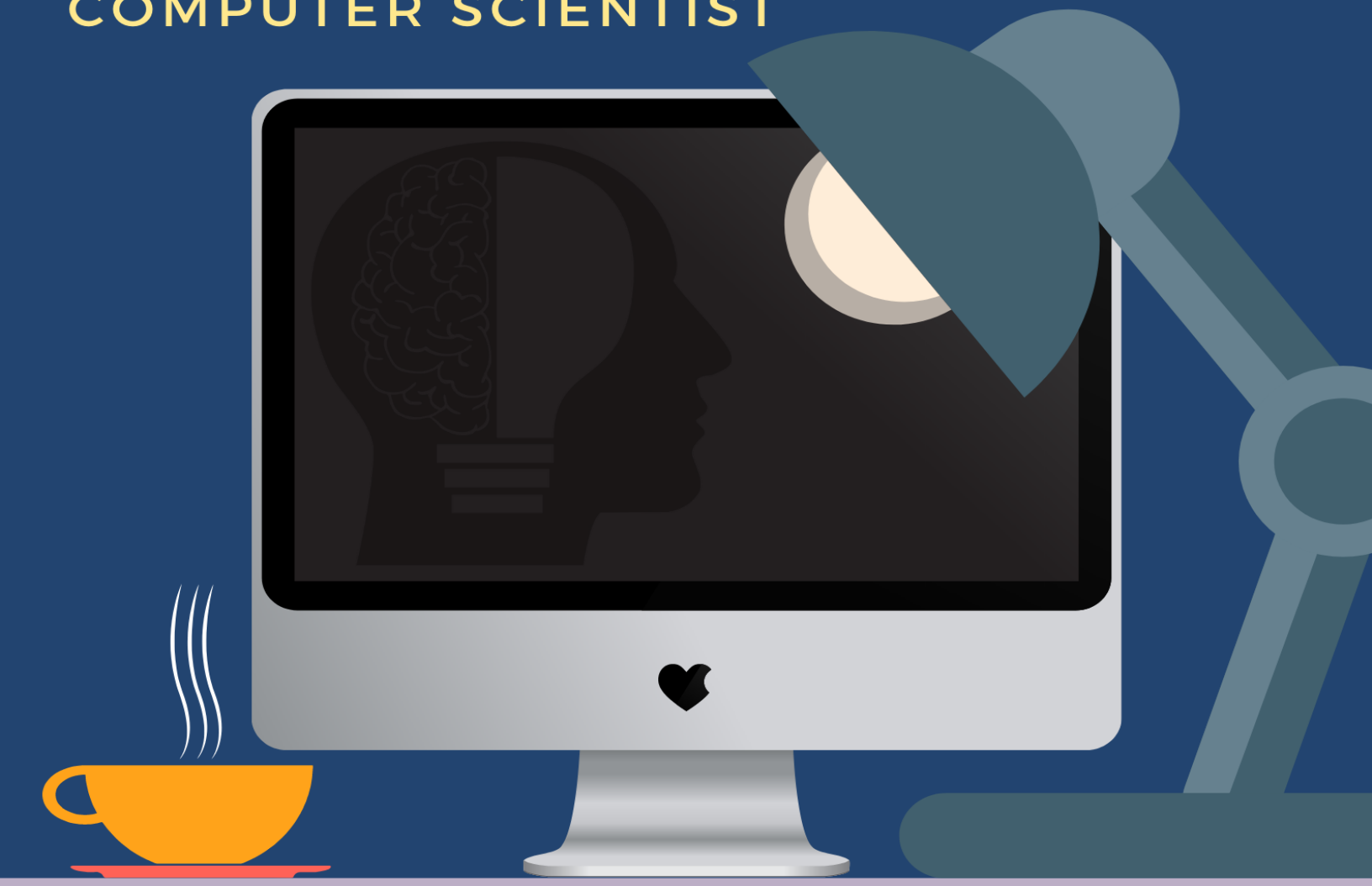
LEARN PYTHON THE RIGHT WAY

HOW TO THINK LIKE A COMPUTER SCIENTIST



Contents

Copyright Notice

Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder why Python? or why teach programming with Python? Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to become frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes. In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming — all of which are applicable to later

courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a higher level of success and a lower level of frustration and that he was able to move faster with better results. Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just programming. I find that using Python allows me to better focus on the actual topic at hand while allowing students to complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction. David Beazley University of Chicago Author of the Python Essential Reference

Preface

By Jeffrey Elkner

This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—never met face to face to work on it, but we have been able to collaborate closely, aided by many other folks who have taken the time and energy to send us their feedback.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

How and why I came to use Python

In 1999, the College Board's Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our GNU/Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free software, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown's talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the Web. I knew that Matt could not have finished an application of that

scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

Finding a textbook

Having decided to use Python in both of my introductory computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free documents came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational materials. Allen had already written a first-year computer science textbook, How to Think Like a Computer Scientist. When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

How to Think Like a Computer Scientist was not just an excellent book, but it had been released under the GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational materials.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the open source community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our Website at http://openbookproject.net called *Python for Fun* and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

Introducing programming with Python

The process of translating and using How to Think Like a Computer Scientist for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional `hello, world` program, which in the Java version of the book looks like this:

```
class Hello {

  public static void main (String[] args) {
      System.out.println ("Hello, world.");
  }
}
```

in the Python version it becomes:

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

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The Java version has always forced me to choose between two unsatisfying options: either to explain the class Hello, public static void main, String[] args„ statements and risk confusing or intimidating some of the students right at the start, or to tell them, Just don't worry about all of that stuff now; we will talk about it later, and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are seven paragraphs of explanation of Hello, world! in the Java version; in the Python version, there are only a few sentences. More importantly, the missing six paragraphs do not deal with the big ideas in computer programming but with the minutia of Java syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put first things first pedagogically. One of the best examples of this is the way in which Python handles variables. In Java a variable is a name for a place that

holds a value if it is a built-in type, and a reference to an object if it is not. Explaining this distinction requires a discussion of how the computer stores data. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of variable that they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply call (type) out its name. Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python improved the effectiveness of our computer science program for all students. I saw a higher general level of success and a lower level of frustration than I experienced teaching with either C++ or Java. I moved faster with better results. More students left the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

Building a community

I have received emails from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion Website at http://openbookproject.net/pybiblio.

With the continued growth of Python, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to me at jeff@elkner.net.

Jeffrey Elkner
Governor's Career and Technical Academy in Arlington
Arlington, Virginia

Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we would like to thank the Free Software Foundation for developing this license and, of course, making it available to us.

We would also like to thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date. It was also getting too large to include everyone who sends in a typo or two. You have our gratitude, and you have the personal satisfaction of making a book you found useful better for you and everyone else who uses it. New additions to the list for the 2nd edition will be those who have made on-going contributions.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, we hope you will take a minute to contact us. The email address (for the Python 3 version of the book) is p.wentworth@ru.ac.za . Substantial changes made due to your suggestions will add you to the next version of the contributor list (unless you ask to be omitted). Thank you!

Second Edition

- An email from Mike MacHenry set me straight on tail recursion. He not only pointed out an error in the presentation, but suggested how to correct it.
- It wasn't until 5th Grade student Owen Davies came to me in a Saturday morning Python enrichment class and said he wanted to write the card game, Gin Rummy, in Python that I finally knew what I wanted to use as the case study for the object oriented programming chapters.
- A special thanks to pioneering students in Jeff's Python Programming class at GCTAA during the 2009-2010 school year: Safath Ahmed, Howard Batiste, Louis Elkner-Alfaro, and Rachel Hancock. Your continual and thoughtfull feedback led to changes in most of the chapters of the book. You set the standard for the active and engaged learners that will help make the new Governor's Academy what it is to become. Thanks to you this is truly a student tested text.
- Thanks in a similar vein to the students in Jeff's Computer Science class at the HB-Woodlawn program during the 2007-2008 school year: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail, and Iliana Vazuka.

- Ammar Nabulsi sent in numerous corrections from Chapters 1 and 2.
- Aldric Giacomoni pointed out an error in our definition of the Fibonacci sequence in Chapter 5.
- Roger Sperberg sent in several spelling corrections and pointed out a twisted piece of logic in Chapter 3.
- Adele Goldberg sat down with Jeff at PyCon 2007 and gave him a list of suggestions and corrections from throughout the book.
- Ben Bruno sent in corrections for chapters 4, 5, 6, and 7.
- Carl LaCombe pointed out that we incorrectly used the term commutative in chapter 6 where symmetric was the correct term.
- Alessandro Montanile sent in corrections for errors in the code examples and text in chapters 3, 12, 15, 17, 18, 19, and 20.
- Emanuele Rusconi found errors in chapters 4, 8, and 15.
- Michael Vogt reported an indentation error in an example in chapter 6, and sent in a suggestion for improving the clarity of the shell vs. script section in chapter 1.

First Edition

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote horsebet.py, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken catTwice function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word unconsciously in Chapter 1 needed to be changed to subconsciously .
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13.

- John Ouzts corrected the definition of return value in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between gleich and selbe.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and is helping us prepare to update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.

Chapter 1: The way of the program

(Watch a video based on this chapter here on YouTube[1].)

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, The way of the program.

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1. The Python programming language

The programming language you will be learning is Python. Python is an example of a high-level language; other **high-level languages** you might have heard of are C++, PHP, Pascal, C#, and Java.

As you might infer from the name high-level language, there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated into something more suitable before they can run.

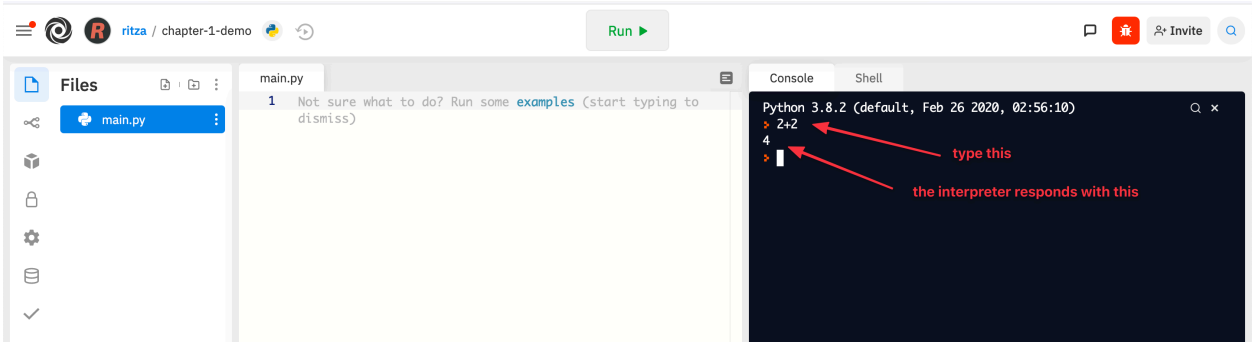
Almost all programs are written in high-level languages because of their advantages. It is much easier to program in a high-level language so programs take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications.

In this edition of the textbook, we use an online programming environment called **Replit**. To follow along with the examples and complete the exercises, all you need is a free account - just navigate to https://replit.com and complete the sign up process.

Once you have an account, create a new repl and choose Python as the language from the dropdown. You'll see it automatically creates a file called `main.py`. By convention, files that contain Python programs have names that end with `.py`.

[1] https://youtu.be/lhtUREG6vAg

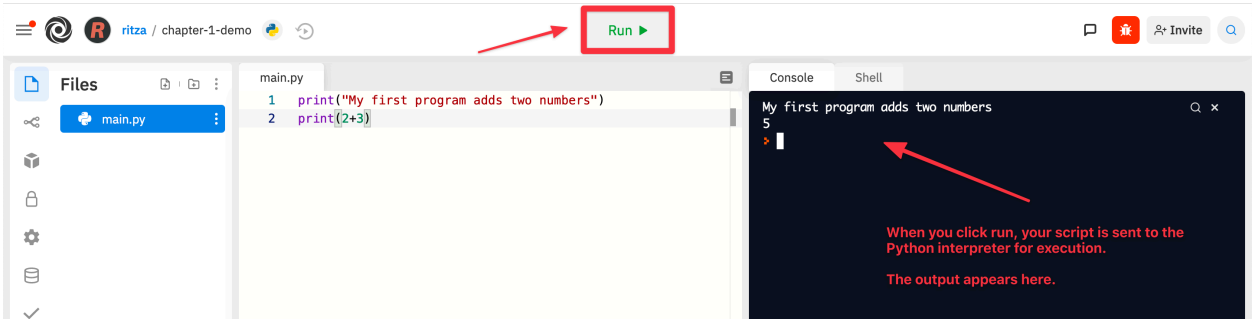
The engine that translates and runs Python is called the **Python Interpreter**: There are two ways to use it: *immediate mode* and *script mode*. In immediate mode, you type Python expressions into the Python Interpreter window, and the interpreter immediately shows the result:



Running code in the interpreter (immediate mode)

The ››› or › is called the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions. We typed `2 + 2`, and the interpreter evaluated our expression, and replied `4`, and on the next line it gave a new prompt, indicating that it is ready for more input.

Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script. Scripts have the advantage that they can be saved to disk, printed, and so on. To create a script, you can enter the code into the middle pane, as shown below



Running code from a file (script mode)

```
1 print("My first program adds two numbers")
2 print(2+3)
```

To execute the program, click the **Run** button in Replit. You're now a computer programmer! Let's take a look at some more theory before we start writing more advanced programs.

1.2. What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input

- Get data from the keyboard, a file, or some other device.

output

- Display data on the screen or send data to a file or other device.

math

- Perform basic mathematical operations like addition and multiplication.

conditional execution

- Check for certain conditions and execute the appropriate sequence of statements.

repetition

- Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with sequences of these basic instructions.

That may be a little vague, but we will come back to this topic later when we talk about **algorithms**.

1.3. What is debugging?

Programming is a complex process, and because it is done by human beings, it often leads to errors. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. Use of the term bug to describe small engineering difficulties dates back to at least 1889, when Thomas Edison had a bug with his phonograph.

Three kinds of errors can occur in a program: syntax errors[2], runtime errors[3], and semantic errors[4].

[2] https://en.wikipedia.org/wiki/Syntax_error
[3] https://en.wikipedia.org/wiki/Runtime_(program_lifecycle_phase)
[4] https://en.wikipedia.org/wiki/Logic_error

It is useful to distinguish between them in order to track them down more quickly.

1.4. Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. Cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.5. Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.6. Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.7. Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system kernel that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus's earlier projects was a program that would switch between displaying AAAA and BBBB. This later evolved to Linux (*The Linux Users' Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

1.8. Formal and natural languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict rules about syntax. For example, `3+3=6` is a syntactically correct mathematical statement, but `3=+6$` is not. `H2O` is a syntactically correct chemical name, but `2Zz` is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, parentheses, commas, and so on. In Python, a statement like `print("Happy New Year for ",2013)` has 6 tokens: a function name, an open parenthesis (round bracket), a string, a comma, a number, and a close parenthesis.

It is possible to make errors in the way one constructs tokens. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as we know). Similarly, `2Zz` is not a legal token in chemistry notation because there is no element with the abbreviation `Zz`.

The second type of syntax rule pertains to the **structure** of a statement— that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before. And in our Python example, if we omitted the comma, or if we changed the two parentheses around to say `print)"Happy New Year for ",2013(` our statement would still have six legal and valid tokens, but the structure is illegal.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell", you understand that the other shoe is the subject and fell is the verb. Once you have parsed a sentence, you can figure out what it means, or the **semantics** of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

ambiguity

- Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy

- In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness

- Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, "The other shoe fell", there is probably no shoe and nothing falling. You'll need to find the original joke to understand the idiomatic meaning of the other shoe falling. Yahoo! Answers thinks it knows!

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

poetry

- Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

prose

- The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

program

- The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.9. The first program

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello, World! In Python, the script looks like this: (For scripts, we'll show line numbers to the left of the Python statements.)

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

This is an example of using the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result shown is

```
1 Hello, World!
```

The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hello, World! program. By this standard, Python does about as well as possible.

1.10. 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.

A **comment** in a computer program is text that is intended only for the human reader — it is completely ignored by the interpreter.

In Python, the `#` token starts a comment. The rest of the line is ignored. Here is a new version of Hello, World!.

```
#-----------------------------------------------
# This demo program shows off how elegant Python is!
# Written by Joe Soap, December 2010.
# Anyone may freely copy or modify this program.
#-----------------------------------------------

print("Hello, World!")     # Isn't this easy!
```

You'll also notice that we've left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse. Use them liberally!

1.11. Glossary

algorithm

A set of specific steps for solving a category of problems.

bug

An error in a program.

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.

debugging

The process of finding and removing any of the three kinds of programming errors.

exception

Another name for a runtime error.

formal language

Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

high-level language

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

immediate mode

A style of using Python where we type expressions at the command prompt, and the results are shown immediately. Contrast with script, and see the entry under Python shell.

interpreter

The engine that executes your Python scripts or expressions.

low-level language

A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

natural language

Any one of the languages that people speak that evolved naturally.

object code

The output of the compiler after it translates the program.

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 function

A function used in a program or script that causes the Python interpreter to display a value on its output device.

problem solving

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

program

a sequence of instructions that specifies to a computer actions and computations to be performed.

Python shell

An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter

for processing. The word shell comes from Unix. In the PyScripter used in this RLE version of the book, the Interpreter Window is where we'd do the immediate mode interaction.

runtime error

An error that does not occur until the program has started to execute but that prevents the program from continuing.

script

A program stored in a file (usually one that will be interpreted).

semantic error

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

semantics

The meaning of a program.

source code

A program in a high-level language before being compiled.

syntax

The structure of a program.

syntax error

An error in a program that makes it impossible to parse — and therefore impossible to interpret.

token

One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

1.12. Exercises

1. Write an English sentence with understandable semantics but incorrect syntax. Write another English sentence which has correct syntax but has semantic errors.
2. Using the Python interpreter, type `1 + 2` and then hit return. Python evaluates this expression, displays the result, and then shows another prompt. `*` is the multiplication operator, and `**` is the exponentiation operator. Experiment by entering different expressions and recording what is displayed by the Python interpreter.
3. Type `1 2` and then hit return. Python tries to evaluate the expression, but it can't because the expression is not syntactically legal. Instead, it shows the error message:

```
  File "<interactive input>", line 1
    1 2
      ^
SyntaxError: invalid syntax
```

In many cases, Python indicates where the syntax error occurred, but it is not always right, and it doesn't give you much information about what is wrong.

So, for the most part, the burden is on you to learn the syntax rules.

In this case, Python is complaining because there is no operator between the numbers.

See if you can find a few more examples of things that will produce error messages when you enter them at the Python prompt. Write down what you enter at the prompt and the last line of the error message that Python reports back to you.

4. Type `print("hello")`. Python executes this, which has the effect of printing the letters h-e-l-l-o. Notice that the quotation marks that you used to enclose the string are not part of the output. Now type `"hello"` and describe your result. Make notes of when you see the quotation marks and when you don't.
5. Type cheese without the quotation marks. The output will look something like this:

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'cheese' is not defined
```

This is a run-time error; specifically, it is a NameError, and even more specifically, it is an error because the name cheese is not defined. If you don't know what that means yet, you will soon.

6. Type `6 + 4 * 9` at the Python prompt and hit enter. Record what happens.

Now create a Python script with the following contents:

```
6 + 4 * 9
```

What happens when you run this script? Now change the script contents to:

```
print(6 + 4 * 9)
```

and run it again.

What happened this time?

Whenever an expression is typed at the Python prompt, it is evaluated and the result is automatically shown on the line below. (Like on your calculator, if you type this expression you'll get the result 42.)

A script is different, however. Evaluations of expressions are not automatically displayed, so it is necessary to use the **print** function to make the answer show up.

It is hardly ever necessary to use the print function in immediate mode at the command prompt.

Chapter 2: Variables, expressions and statements

(Watch a video based on this chapter here on YouTube[5].)

2.1. Values and data types

A **value** is one of the fundamental things — like a letter or a number — that a program manipulates. The values we have seen so far are `4` (the result when we added `2 + 2`), and `"Hello, World!"`.

These values are classified into different **classes**, or **data types**: `4` 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.

If you are not sure what class a value falls into, Python has a function called **type** which can tell you.

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

Not surprisingly, strings belong to the class **str** and integers belong to the class **int**. Less obviously, numbers with a decimal point belong to a class called **float**, because these numbers are represented in a format called *floating-point*. At this stage, you can treat the words *class* and *type* interchangeably. We'll come back to a deeper understanding of what a class is in later chapters.

```
>>> 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.

[5] https://youtu.be/gIvstR16coI

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

They're strings!

Strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (''' or """)

```
>>> type('This is a string.')
<class 'str'>
>>> type("And so is this.")
<class 'str'>
>>> type("""and this.""")
<class 'str'>
>>> type('''and even this...''')
<class 'str'>
```

Double quoted strings can contain single quotes inside them, as in `"Bruce's beard"`, and single quoted strings can have double quotes inside them, as in `'The knights who say "Ni!"'`.

Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
>>> print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
```

Triple quoted strings can even span multiple lines:

```
>>> message = """This message will
... span several
... lines."""
>>> print(message)
This message will
span several
lines.
>>>
```

Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings: once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value. But when the interpreter wants to display a string, it has to decide which quotes to use to make it look like a string.

```
>>> 'This is a string.'
'This is a string.'
>>> """And so is this."""
'And so is this.'
```

So the Python language designers usually chose to surround their strings by single quotes. What do you think would happen if the string already contained single quotes?

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `42,000`. This is not a legal integer in Python, but it does mean something else, which is legal:

```
>>> 42000
42000
>>> 42,000
(42, 0)
```

Well, that's not what we expected at all! Because of the comma, Python chose to treat this as a pair of values. We'll come back to learn about pairs later. But, for the moment, remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the previous chapter: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intended.

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.

The **assignment statement** gives a value to a variable:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

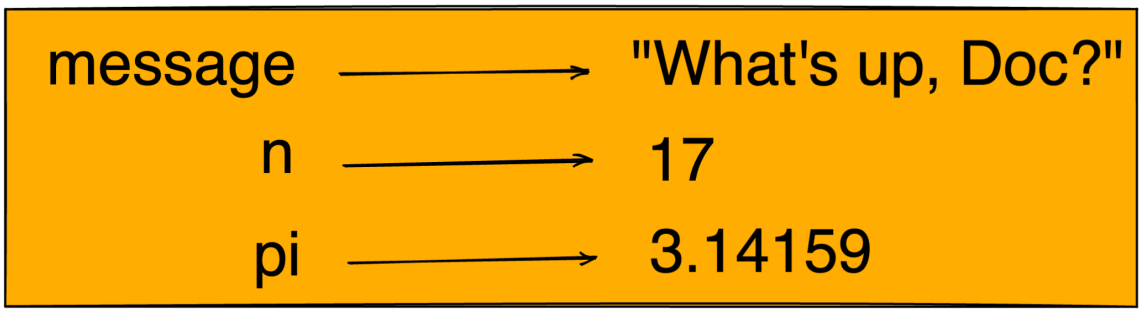
This example makes three assignments. The first assigns the string value `"What's up, Doc?"` to a variable named message. The second gives the integer `17` to `n`, and the third assigns the floating-point number `3.14159` to a variable called `pi`.

The assignment token, `=`, should not be confused with *equals*, which uses the token `==`. The assignment statement binds a *name*, on the left-hand side of the operator, to a *value*, on the right-hand side. This is why you will get an error if you enter:

```
>>> 17 = n
File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

Tip:
When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17". Don't say "n equals 17".

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind). This diagram shows the result of executing the assignment statements:



State Snapshot

If you ask the interpreter to evaluate a variable, it will produce the value that is currently linked to the variable:

```
>>> message
"What's up, Doc?"
>>> n
17
>>> pi
3.14159
```

We use variables in a program to "remember" things, perhaps the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. (*This is different from maths. In maths, if you give* `x` *the value* `3`*, it cannot change to link to a different value half-way through your calculations!*)

```
>>> day = "Thursday"
>>> day
'Thursday'
>>> day = "Friday"
>>> day
'Friday'
>>> day = 21
>>> day
21
```

You'll notice we changed the value of `day` three times, and on the third assignment we even made it refer to a value that was of a different type.

A great deal of programming is about having the computer remember things, e.g. *The number of missed calls on your phone*, and then arranging to update or change the variable when you miss another call.

2.3. Variable names and keywords

Variable names can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `price_of_tea_in_china`.

There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

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 = "Computer Science 101"
SyntaxError: invalid syntax
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that `class` is one of the Python **keywords**. Keywords define the language's syntax rules and structure, and they cannot be used as variable names.

Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

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

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.

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for.

Caution

Beginners sometimes confuse "meaningful to the human readers" with "meaningful to the computer". So they'll wrongly think that because they've called some variable `average` *or* `pi`*, it will somehow magically calculate an average, or magically know that the variable* `pi` *should have a value like* `3.14159`*. No! The computer doesn't understand what you intend the variable to mean.*

So you'll find some instructors who deliberately don't choose meaningful names when they teach beginners — not because we don't think it is a good habit, but because we're trying to reinforce the message that you — the programmer — must write the program code to calculate the average, and you must write an assignment statement to give the variable `pi` *the value you want it to have.*

2.4. Statements

A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we'll see shortly are `while` statements, `for` statements, `if` statements, and `import` statements. (There are other kinds too!)

When you type a statement on the command line, Python executes it. Statements don't produce any result.

2.5. Evaluating expressions

An **expression** is a combination of values, variables, operators, and calls to functions. If you type an expression at the Python prompt, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
>>> len("hello")
5
```

In this example `len` is a built-in Python function that returns the number of characters in a `string`. We've previously seen the `print` and the `type` functions, so this is our third example of a function!

The *evaluation* of an *expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> y = 3.14
>>> x = len("hello")
>>> x
5
>>> y
3.14
```

2.6. Operators and operands

Operators are special tokens that represent computations like addition, multiplication and division. The values the operator uses are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

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

The tokens `+`, `-`, and `*`, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (`*`) is the token for multiplication, and `**` is the token for exponentiation.

```
>>> 2 ** 3
8
>>> 3 ** 2
9
```

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect.

Example: so let us convert 645 minutes into hours:

```
>>> minutes = 645
>>> hours = minutes / 60
>>> hours
10.75
```

Oops! In Python 3, the division operator `/` always yields a floating point result. What we might have wanted to know was how many whole hours there are, and how many minutes remain. Python gives us two different flavors of the division operator. The second, called **floor division** uses the token `//`. Its result is always a whole number — and if it has to adjust the number it always moves it to the left on the number line. So `6 // 4` yields `1`, but `-6 // 4` might surprise you!

```
>>> 7 / 4
1.75
>>> 7 // 4
1
>>> minutes = 645
>>> hours = minutes // 60
>>> hours
10
```

Take care that you choose the correct flavor of the division operator. If you're working with expressions where you need floating point values, use the division operator that does the division accurately.

2.7. Type converter functions

Here we'll look at three more Python functions, `int`, `float` and `str`, which will (attempt to) convert their arguments into types `int`, `float` and `str` respectively. We call these **type converter** functions.

The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number — a process we call truncation towards zero on the number line. Let us see this in action:

```
>>> int(3.14)
3
>>> int(3.9999)            # This doesn't round to the closest int!
3
>>> int(3.0)
3
>>> int(-3.999)            # Note that the result is closer to zero
-3
>>> int(minutes / 60)
10
>>> int("2345")            # Parse a string to produce an int
2345
>>> int(17)                # It even works if arg is already an int
17
>>> int("23 bottles")
```

This last case doesn't look like a number — what do we expect?

```
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 bottles'
```

The type converter `float` can turn an integer, a float, or a syntactically legal string into a float:

```
>>> float(17)
17.0
>>> float("123.45")
123.45
```

The type converter `str` turns its argument into a string:

```
>>> str(17)
'17'
>>> str(123.45)
'123.45'
```

2.8. Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

1. **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 though it doesn't change the result.
2. **E**xponentiation has the next highest precedence, so `2**1+1` is `3` and not `4`, and `3*1**3` is `3` and not `27`.
3. **M**ultiplication and both **D**ivision operators have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` yields `5` rather than `4`, and `5-2*2` is `1`, not `6`.

Operators with the same precedence are evaluated from left-to-right. In algebra we say they are left-associative. So in the expression `6-3+2`, the subtraction happens first, yielding `3`. We then add `2` to get the result `5`. If the operations had been evaluated from right to left, the result would have been `6-(3+2)`, which is `1`. (The acronym PEDMAS could mislead you to thinking that division has higher precedence than multiplication, and addition is done ahead of subtraction - don't be misled. Subtraction and addition are at the same precedence, and the left-to-right rule applies.)

Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator `**`, so a useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```
>>> 2 ** 3 ** 2     # The right-most ** operator gets done first!
512
>>> (2 ** 3) ** 2   # Use parentheses to force the order you want!
64
```

The immediate mode command prompt of Python is great for exploring and experimenting with expressions like this.

2.9. Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
>>> message - 1        # Error
>>> "Hello" / 123      # Error
>>> message * "Hello"  # Error
>>> "15" + 2           # Error
```

Interestingly, the `+` operator does work with strings, but for strings, the `+` operator represents **concatenation**, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
baked_good = " nut bread"
print(fruit + baked_good)
```

The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.

The * operator also works on strings; it performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string; the other has to be an integer.

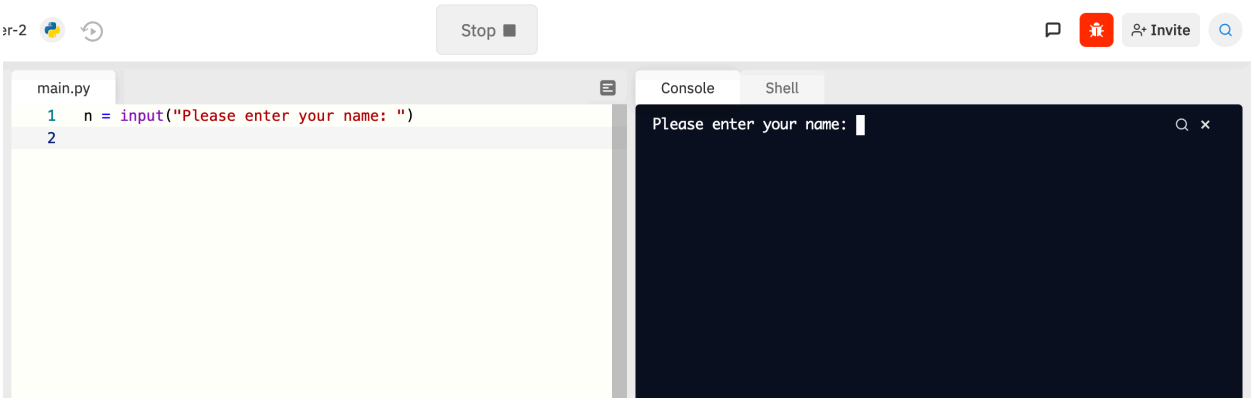
On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `"Fun"*3` to be the same as `"Fun"+"Fun"+"Fun"`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

2.10. Input

There is a built-in function in Python for getting input from the user:

```
n = input("Please enter your name: ")
```

A sample run of this script in Replit would populate your input question in the console to the left like this:



Input Prompt

The user of the program can enter the name and press enter, and when this happens the text that has been entered is returned from the input function, and in this case assigned to the variable n.

Even if you asked the user to enter their age, you would get back a string like `"17"`. It would be your job, as the programmer, to convert that string into a int or a float, using the `int` or `float` converter functions we saw earlier.

2.11. Composition

So far, we have looked at the elements of a program — variables, expressions, statements, and function calls — in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them into larger chunks.

For example, we know how to get the user to enter some input, we know how to convert the string we get into a float, we know how to write a complex expression, and we know how to print values. Let's put these together in a small four-step program that asks the user to input a value for the radius of a circle, and then computes the area of the circle from the formula

$$A = \pi r^2$$

Area of a circle

Firstly, we'll do the four steps one at a time:

```
response = input("What is your radius? ")
r = float(response)
area = 3.14159 * r**2
print("The area is ", area)
```

Now let's compose the first two lines into a single line of code, and compose the second two lines into another line of code.

```
r = float( input("What is your radius? ") )
print("The area is ", 3.14159 * r**2)
```

If we really wanted to be tricky, we could write it all in one statement:

```
print("The area is ", 3.14159*float(input("What is your radius?"))**2)
```

Such compact code may not be most understandable for humans, but it does illustrate how we can compose bigger chunks from our building blocks.

If you're ever in doubt about whether to compose code or fragment it into smaller steps, try to make it as simple as you can for the human to follow. My choice would be the first case above, with four separate steps.

2.12. The modulus operator

The modulus operator works on integers (and integer expressions) and gives the remainder when the first number is divided by the second. In Python, the modulus operator is a percent sign (`%`). The syntax is the same as for other operators. It has the same precedence as the multiplication operator.

```
>>> q = 7 // 3     # This is integer division operator
>>> print(q)
2
>>> r  = 7 % 3
>>> print(r)
1
```

So `7` divided by `3` is `2` with a remainder of `1`.

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`.

Also, you can 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.

It is also extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. So let's write a program to ask the user to enter some seconds, and we'll convert them into hours, minutes, and remaining seconds.

```
total_secs = int(input("How many seconds, in total?"))
hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60

print("Hrs=", hours, "  mins=", minutes,
                        "secs=", secs_finally_remaining)
```

2.13. Glossary

assignment statement

A statement that assigns a value to a name (variable). To the left of the assignment operator, =, is a name. To the right of the assignment token is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the following assignment:

```
1 n = n + 1
```

`n` plays a very different role on each side of the =. On the right it is a value and makes up part of the expression which will be evaluated by the Python interpreter before assigning it to the name on the left.

assignment token

= is Python's assignment token. Do not confuse it with *equals*, which is an operator for comparing values.

composition

The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

concatenate

To join two strings end-to-end.

data type

A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (`int`), floating-point numbers (`float`), and strings (`str`).

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.

float

A Python data type which stores *floating-point* numbers. Floating-point numbers are stored internally in two parts: a *base* and an *exponent*. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use `floats`, and remember that they are only approximate values.

floor division

An operator (denoted by the token `//`) that divides one number by another and yields an integer, or, if the result is not already an integer, it yields the next smallest integer.

int

A Python data type that holds positive and negative whole numbers.

keyword

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

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.

state snapshot

A graphical representation of a set of variables and the values to which they refer, taken at a particular instant during the program's execution.

statement

An instruction that the Python interpreter can execute. So far we have only seen the assignment statement, but we will soon meet the `import` statement and the `for` statement.

str

A Python data type that holds a string of characters.

value

A number or string (or other things to be named later) that can be stored in a variable or computed in an expression.

variable

A name that refers to a value.

variable name

A name given to a variable. Variable names in Python consist of a sequence of letters (`a..z`, `A..Z`, and _) and digits (0..9) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program *self documenting*.

2.14. Exercises

1. Take the sentence: All work and no play makes Jack a dull boy. Store each word in a separate variable, then print out the sentence on one line using print.
2. Add parenthesis to the expression `6 * 1 - 2` to change its value from `4` to `-6`.
3. Place a comment before a line of code that previously worked, and record what happens when you rerun the program.
4. Start the Python interpreter and enter `bruce + 4` at the prompt. This will give you an error:

```
NameError: name 'bruce' is not defined
```

   Assign a value to bruce so that `bruce + 4` evaluates to `10`.
5. The formula for computing the final amount if one is earning compound interest is given on Wikipedia as

   Compounded Interest Formula

$$P' = P\left(1 + \frac{r}{n}\right)^{nt}$$

where:

P is the original principal sum

P′ is the new principal sum

r is the nominal annual interest rate

n is the compounding frequency

t is the overall length of time the interest is applied (expressed using the same time units as *r*, usually years).

Compounded Interest Formula

   Write a Python program that assigns the principal amount of $10000 to variable `P`, assign to n the value `12`, and assign to `r` the interest rate of `8%`. Then have the program prompt the user for the number of years `t` that the money will be compounded for. Calculate and print the final amount after `t` years.
6. Evaluate the following numerical expressions in your head, then use the Python interpreter to check your results:

```
>>> 5 % 2
>>> 9 % 5
>>> 15 % 12
>>> 12 % 15
>>> 6 % 6
>>> 0 % 7
>>> 7 % 0
```

   What happened with the last example? Why? If you were able to correctly anticipate the computer's response in all but the last one, it is time to move on. If not, take time now to

make up examples of your own. Explore the modulus operator until you are confident you understand how it works.

7. You look at the clock and it is exactly 2pm. You set an alarm to go off in 51 hours. At what time does the alarm go off? (*Hint: you could count on your fingers, but this is not what we're after. If you are tempted to count on your fingers, change the 51 to 5100.*)
8. Write a Python program to solve the general version of the above problem. Ask the user for the time now (in hours), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off.

Chapter 3: Hello, little turtles!

There are many *modules* in Python that provide very powerful features that we can use in our own programs. Some of these can send email, or fetch web pages. The one we'll look at in this chapter allows us to create turtles and get them to draw shapes and patterns.

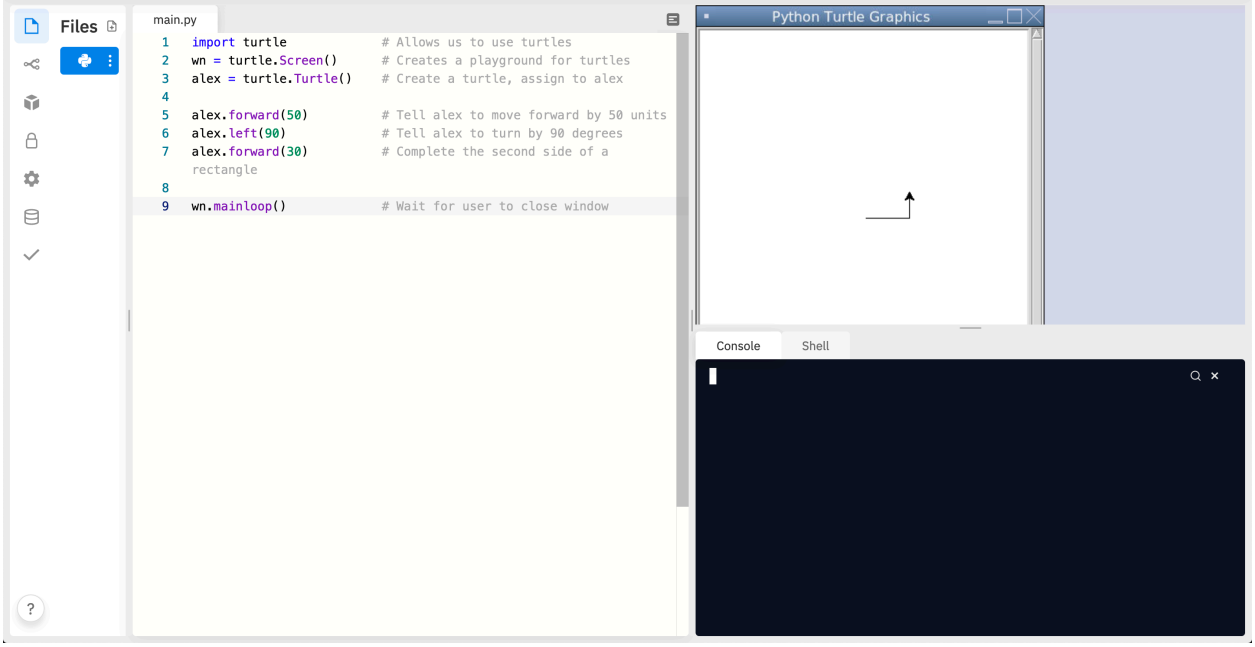
The turtles are fun, but the real purpose of the chapter is to teach ourselves a little more Python, and to develop our theme of *computational thinking*, or *thinking like a computer scientist*. Most of the Python covered here will be explored in more depth later.

3.1. Our first turtle program

Let's write a couple of lines of Python program to create a new turtle and start drawing a rectangle. (We'll call the variable that refers to our first turtle `alex`, but we can choose another name if we follow the naming rules from the previous chapter).

```
1  import turtle              # Allows us to use turtles
2  wn = turtle.Screen()       # Creates a playground for turtles
3  alex = turtle.Turtle()     # Create a turtle, assign to alex
4
5  alex.forward(50)           # Tell alex to move forward by 50 units
6  alex.left(90)              # Tell alex to turn by 90 degrees
7  alex.forward(30)           # Complete the second side of a rectangle
8
9  wn.mainloop()              # Wait for user to close window
```

When we run this program, a new window pops up:



Turtle Window

Here are a couple of things we'll need to understand about this program.

The first line tells Python to load a module named `turtle`. That module brings us two new types that we can use: the `Turtle` type, and the `Screen` type. The dot notation `turtle.Turtle` means "*The Turtle type that is defined within the turtle module*". (Remember that Python is case sensitive, so the module name, with a lowercase `t`, is different from the type `Turtle`.)

We then create and open what it calls a screen (we would prefer to call it a window), which we assign to variable `wn`. Every window contains a **canvas**, which is the area inside the window on which we can draw.

In line 3 we create a turtle. The variable `alex` is made to refer to this turtle.

So these first three lines have set things up, we're ready to get our turtle to draw on our canvas.

In lines 5-7, we instruct the **object** `alex` to move, and to turn. We do this by **invoking**, or activating, `alex's` **methods** — these are the instructions that all turtles know how to respond to.

The last line plays a part too: the `wn` variable refers to the window shown above. When we invoke its `mainloop` method, it enters a state where it waits for events (like keypresses, or mouse movement and clicks). The program will terminate when the user closes the window.

An object can have various methods — things it can do — and it can also have **attributes** — (sometimes called properties). For example, each turtle has a *color* attribute. The method invocation `alex.color("red")` will make `alex` red, and drawing will be red too. (Note the word *color* is spelled the American way!)

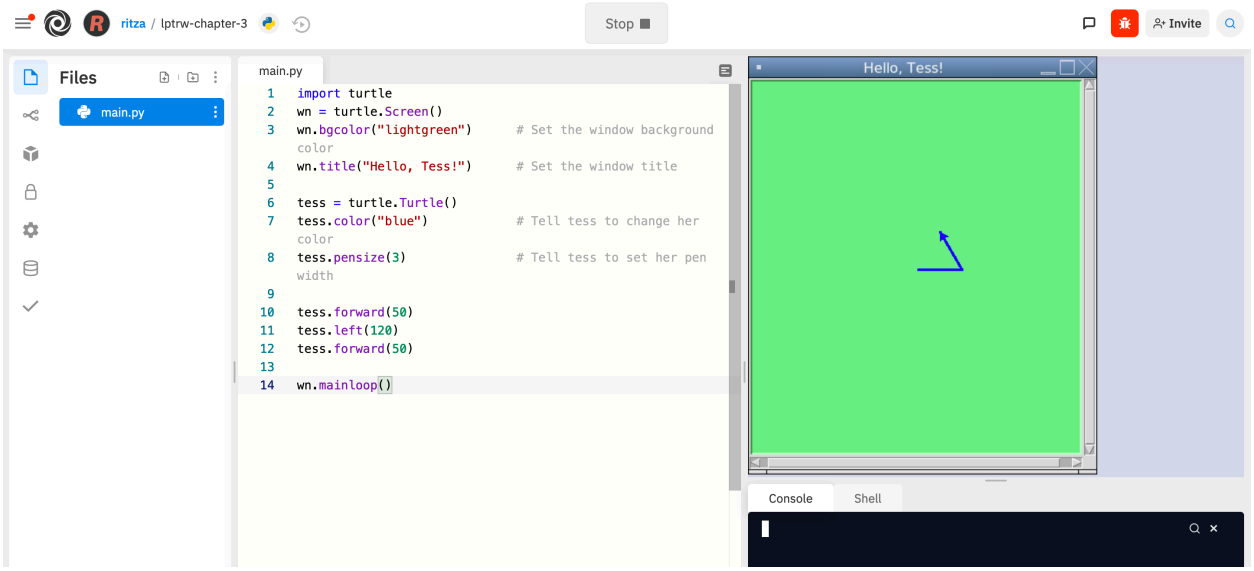
The color of the turtle, the width of its pen, the position of the turtle within the window, which way it is facing, and so on are all part of its current **state**. Similarly, the window object has a background

color, and some text in the title bar, and a size and position on the screen. These are all part of the state of the window object.

Quite a number of methods exist that allow us to modify the turtle and the window objects. We'll just show a couple. In this program we've only commented those lines that are different from the previous example (and we've used a different variable name for this turtle):

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")      # Set the window background color
wn.title("Hello, Tess!")      # Set the window title

tess = turtle.Turtle()
tess.color("blue")            # Tell tess to change her color
tess.pensize(3)               # Tell tess to set her pen width

tess.forward(50)
tess.left(120)
tess.forward(50)

wn.mainloop()
```

When we run this program, this new window pops up, and will remain on the screen until we close it.



tess mainloop

Extend this program ...

1. Modify this program so that before it creates the window, it prompts the user to enter the desired background color. It should store the user's responses in a variable, and modify the color of the window according to the user's wishes. (*Hint: you can find a list of permitted color names at http://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm. It includes some quite unusual ones, like "peach puff" and "HotPink".*)
2. Do similar changes to allow the user, at runtime, to set tess' color.
3. Do the same for the width of `tess'` pen. *Hint: your dialog with the user will return a string, but tess'* `pensize` *method expects its argument to be an int. So you'll need to convert the string to an int before you pass it to* `pensize`.

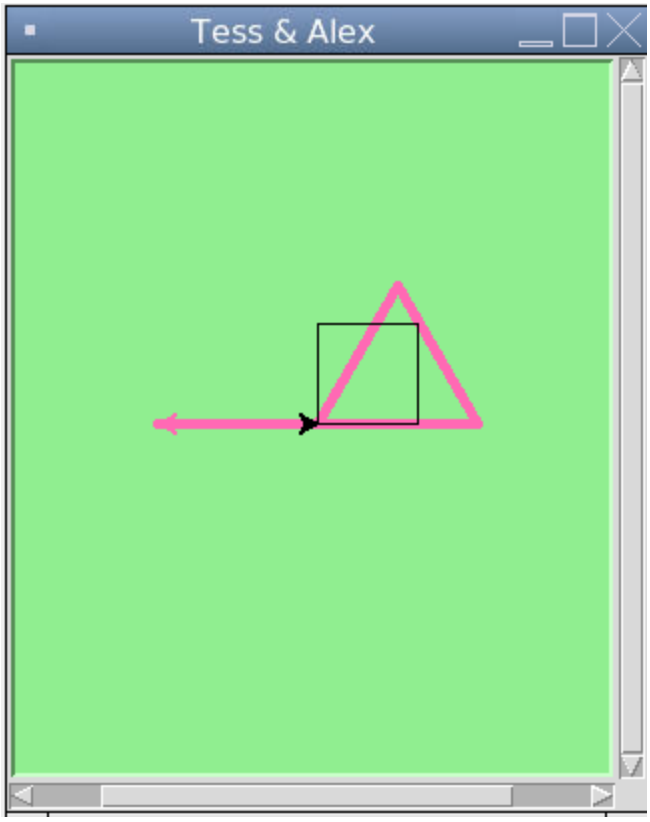
3.2. Instances — a herd of turtles

Just like we can have many different integers in a program, we can have many turtles. Each of them is called an **instance**. Each instance has its own attributes and methods — so alex might draw with a thin black pen and be at some position, while `tess` might be going in her own direction with a fat pink pen.

```
import turtle
wn = turtle.Screen()         # Set up the window and its attributes
wn.bgcolor("lightgreen")
wn.title("Tess & Alex")

tess = turtle.Turtle()       # Create tess and set some attributes
tess.color("hotpink")
tess.pensize(5)

alex = turtle.Turtle()       # Create alex

tess.forward(80)             # Make tess draw equilateral triangle
tess.left(120)
tess.forward(80)
tess.left(120)
tess.forward(80)
tess.left(120)               # Complete the triangle

tess.right(180)              # Turn tess around
tess.forward(80)             # Move her away from the origin

alex.forward(50)             # Make alex draw a square
alex.left(90)
alex.forward(50)
alex.left(90)
```

```
26 alex.forward(50)
27 alex.left(90)
28 alex.forward(50)
29 alex.left(90)
30
31 wn.mainloop()
```

Here is what happens when `alex` completes his rectangle, and `tess` completes her triangle:



Alex and Tess

Here are some *How to think like a computer scientist* observations:

- There are 360 degrees in a full circle. If we add up all the turns that a turtle makes, no matter what steps occurred between the turns, we can easily figure out if they add up to some multiple of 360. This should convince us that alex is facing in exactly the same direction as he was when he was first created. (Geometry conventions have 0 degrees facing East, and that is the case here too!)
- We could have left out the last turn for alex, but that would not have been as satisfying. If we're asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in. This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!

- We did the same with tess: she drew her triangle, and turned through a full 360 degrees. Then we turned her around and moved her aside. Even the blank line 18 is a hint about how the programmer's mental chunking is working: in big terms, tess' movements were chunked as "draw the triangle" (lines 12-17) and then "move away from the origin" (lines 19 and 20).
- One of the key uses for comments is to record our mental chunking, and big ideas. They're not always explicit in the code.
- And, uh-huh, two turtles may not be enough for a herd. But the important idea is that the turtle module gives us a kind of factory that lets us create as many turtles as we need. Each instance has its own state and behaviour.

3.3. The for loop

When we drew the square, it was quite tedious. We had to explicitly repeat the steps of moving and turning four times. If we were drawing a hexagon, or an octagon, or a polygon with 42 sides, it would have been worse.

So a basic building block of all programs is to be able to repeat some code, over and over again.

Python's **`for`** loop solves this for us. Let's say we have some friends, and we'd like to send them each an email inviting them to our party. We don't quite know how to send email yet, so for the moment we'll just print a message for each friend:

```
1 for f in ["Joe","Zoe","Brad","Angelina","Zuki","Thandi","Paris"]:
2     invite = "Hi " + f + ".  Please come to my party on Saturday!"
3     print(invite)
4 # more code can follow here ...
```

When we run this, the output looks like this:

```
1 Hi Joe.  Please come to my party on Saturday!
2 Hi Zoe.  Please come to my party on Saturday!
3 Hi Brad.  Please come to my party on Saturday!
4 Hi Angelina.  Please come to my party on Saturday!
5 Hi Zuki.  Please come to my party on Saturday!
6 Hi Thandi.  Please come to my party on Saturday!
7 Hi Paris.  Please come to my party on Saturday!
```

- The variable `f` in the for statement at line `1` is called the **loop variable**. We could have chosen any other variable name instead.

Lines `2` and `3` are the **loop body**. The loop body is always indented. The indentation determines exactly what statements are "in the body of the loop".

- On each *iteration* or *pass* of the loop, first a check is done to see if there are still more items to be processed. If there are none left (this is called the **terminating condition** of the loop), the loop has finished. Program execution continues at the next statement after the loop body, (e.g. in this case the next statement below the comment in line `4`).
- If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here `7` times, and each time `f` will refer to a different friend.
- At the end of each execution of the body of the loop, Python returns to the for statement, to see if there are more items to be handled, and to assign the next one to `f`.

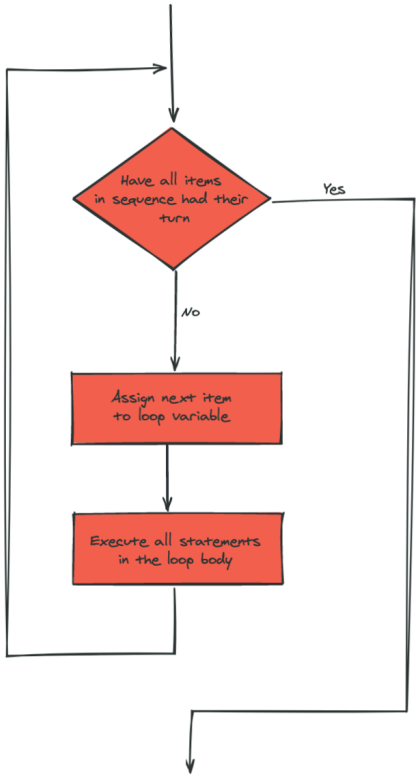
3.4. Flow of Execution of the for loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, of the **flow of execution** of the program. When humans execute programs, they often use their finger to point to each statement in turn. So we could think of control flow as "Python's moving finger".

Control flow until now has been strictly top to bottom, one statement at a time. The for loop changes this.

Flowchart of a for loop

Control flow is often easy to visualize and understand if we draw a flowchart. This shows the exact steps and logic of how the for statement executes.



For loop flowchart

3.5. The loop simplifies our turtle program

To draw a square we'd like to do the same thing four times — move the turtle, and turn. We previously used 8 lines to have alex draw the four sides of a square. This does exactly the same, but using just three lines:

```
for i in [0,1,2,3]:
    alex.forward(50)
    alex.left(90)
```

Some observations:

- While "saving some lines of code" might be convenient, it is not the big deal here. What is much more important is that we've found a "repeating pattern" of statements, and reorganized our program to repeat the pattern. Finding the chunks and somehow getting our programs arranged around those chunks is a vital skill in computational thinking.

- The values `[0,1,2,3]` were provided to make the loop body execute `4` times. We could have used any four values, but these are the conventional ones to use. In fact, they are so popular that Python gives us special built-in range objects:

```
for i in range(4):
    # Executes the body with i = 0, then 1, then 2, then 3
for x in range(10):
    # Sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Computer scientists like to count from `0`!
- `range` can deliver a sequence of values to the loop variable in the `for` loop. They start at `0`, and in these cases do not include the `4` or the `10`.
- Our little trick earlier to make sure that alex did the final turn to complete 360 degrees has paid off: if we had not done that, then we would not have been able to use a loop for the fourth side of the square. It would have become a "special case", different from the other sides. When possible, we'd much prefer to make our code fit a general pattern, rather than have to create a special case.

So to repeat something four times, a good Python programmer would do this:

```
for i in range(4):
    alex.forward(50)
    alex.left(90)
```

By now you should be able to see how to change our previous program so that tess can also use a `for` loop to draw her equilateral triangle.

But now, what would happen if we made this change?

```
for c in ["yellow", "red", "purple", "blue"]:
    alex.color(c)
    alex.forward(50)
    alex.left(90)
```

A variable can also be assigned a value that is a list. So lists can also be used in more general situations, not only in the `for` loop. The code above could be rewritten like this:

```
# Assign a list to a variable
clrs = ["yellow", "red", "purple", "blue"]
for c in clrs:
    alex.color(c)
    alex.forward(50)
    alex.left(90)
```

3.6. A few more turtle methods and tricks

Turtle methods can use negative angles or distances. So `tess.forward(-100)` will move tess backwards, and `tess.left(-30)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will get tess facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

This suggests that we don't need both a left and a right turn method — we could be minimalists, and just have one method. There is also a *backward* method. (If you are very nerdy, you might enjoy saying `alex.backward(-100)` to move `alex` forward!)

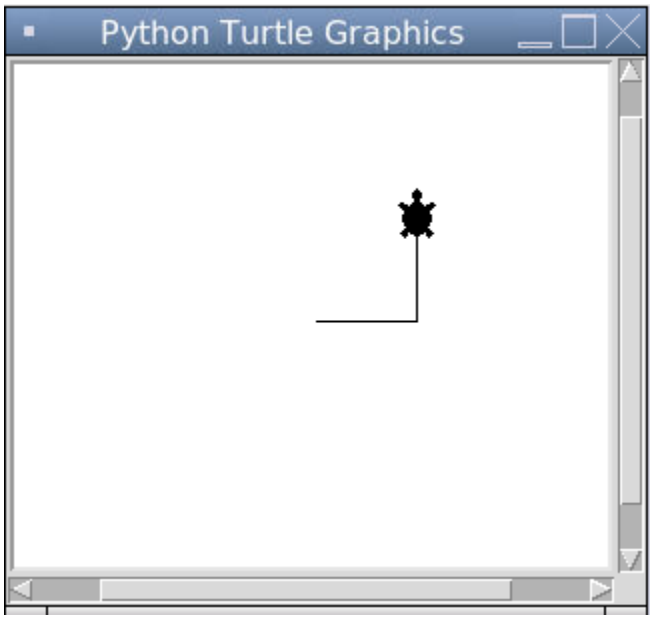
Part of thinking like a scientist is to understand more of the structure and rich relationships in our field. So revising a few basic facts about geometry and number lines, and spotting the relationships between left, right, backward, forward, negative and positive distances or angles values is a good start if we're going to play with turtles.

A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The methods are

```
alex.penup()
alex.forward(100)     # This moves alex, but no line is drawn
alex.pendown()
```

Every turtle can have its own shape. The ones available "out of the box" are `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
alex.shape("turtle")
```



Turtle Shape

We can speed up or slow down the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between `1` (slowest) to `10` (fastest). But if we set the speed to `0`, it has a special meaning — turn off animation and go as fast as possible.

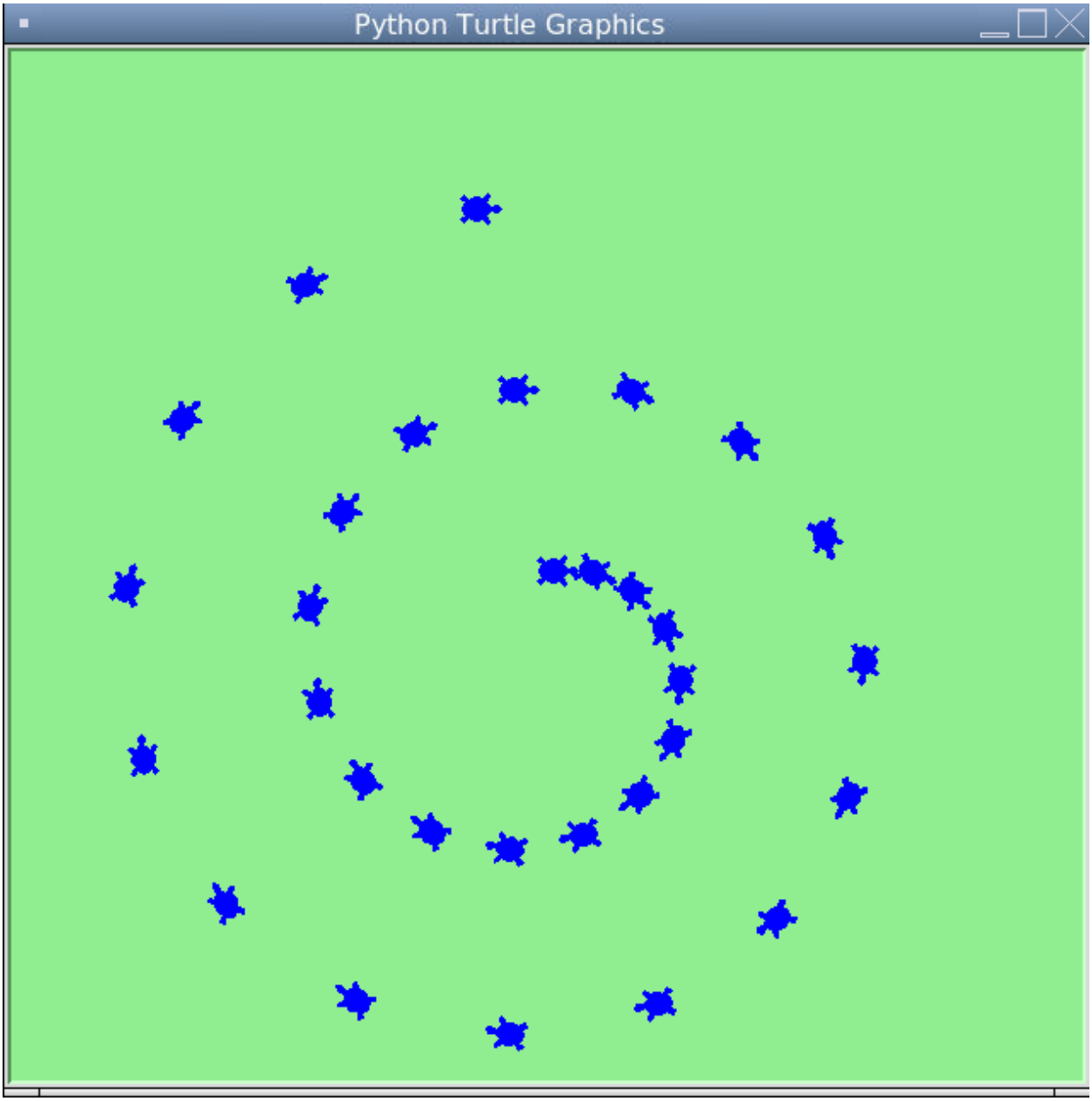
```
alex.speed(10)
```

A turtle can "stamp" its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works, even when the pen is up.

Let's do an example that shows off some of these new features:

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")
tess = turtle.Turtle()
tess.shape("turtle")
tess.color("blue")

tess.penup()                # This is new
size = 20
for i in range(30):
    tess.stamp()            # Leave an impression on the canvas
    size = size + 3         # Increase the size on every iteration
    tess.forward(size)      # Move tess along
    tess.right(24)          #  ...  and turn her
```

```
15
16 wn.mainloop()
```



Turtle Spiral

Be careful now! How many times was the body of the loop executed? How many turtle images do we see on the screen? All except one of the shapes we see on the screen here are footprints created by stamp. But the program still only has one turtle instance — can you figure out which one here is the real tess? (*Hint: if you're not sure, write a new line of code after the for loop to change tess' color, or to put her pen down and draw a line, or to change her shape, etc.*)

3.7. Glossary

attribute

Some state or value that belongs to a particular object. For example, tess has a color.

canvas

A surface within a window where drawing takes place.

control flow

See flow of execution in the next chapter.

for loop

A statement in Python for convenient repetition of statements in the body of the loop.

loop body

Any number of statements nested inside a loop. The nesting is indicated by the fact that the statements are indented under the `for` loop statement.

loop variable

A variable used as part of a `for` loop. It is assigned a different value on each iteration of the loop.

instance

An object of a certain type, or class. tess and alex are different instances of the class Turtle.

method

A function that is attached to an object. Invoking or activating the method causes the object to respond in some way, e.g. forward is the method when we say `tess.forward(100)`.

invoke

An object has methods. We use the verb invoke to mean activate the method. Invoking a method is done by putting parentheses after the method name, with some possible arguments. So `tess.forward()` is an invocation of the forward method.

module

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the import statement.

object

A "thing" to which a variable can refer. This could be a screen window, or one of the turtles we have created.

range

A built-in function in Python for generating sequences of integers. It is especially useful when we need to write a for loop that executes a fixed number of times.

terminating condition

A condition that occurs which causes a loop to stop repeating its body. In the for loops we saw in this chapter, the terminating condition has been when there are no more elements to assign to the loop variable.

3.8. Exercises

1. Write a program that prints `We like Python's turtles!` 1000 times.
2. Give three attributes of your cellphone object. Give three methods of your cellphone.
3. Write a program that uses a `for` loop to print

```
One of the months of the year is January
One of the months of the year is February
...
```

4. Suppose our turtle tess is at heading `0` — facing east. We execute the statement `tess.left(3645)`. What does tess do, and what is her final heading?
5. Assume you have the assignment `xs = [12, 10, 32, 3, 66, 17, 42, 99, 20]`

   a. Write a loop that prints each of the numbers on a new line.

   b. Write a loop that prints each number and its square on a new line.

   c. Write a loop that adds all the numbers from the list into a variable called `total`. You should set the `total` variable to have the value `0` before you start adding them up, and print the value in total after the loop has completed.

   d. Print the product of all the numbers in the list. (product means all multiplied together)
6. Use `for` loops to make a turtle draw these regular polygons (regular means all sides the same lengths, all angles the same):

   - An equilateral triangle
   - A square
   - A hexagon (six sides)
   - An octagon (eight sides)
7. A drunk pirate makes a random turn and then takes `100` steps forward, makes another random turn, takes another `100` steps, turns another random amount, etc. A social science student records the angle of each turn before the next `100` steps are taken. Her experimental data is `[160, -43, 270, -97, -43, 200, -940, 17, -86]`. (Positive angles are counter-clockwise.) Use a turtle to draw the path taken by our drunk friend.
8. Enhance your program above to also tell us what the drunk pirate's heading is after he has finished stumbling around. (Assume he begins at heading `0`).
9. If you were going to draw a regular polygon with 18 sides, what angle would you need to turn the turtle at each corner?
10. At the interactive prompt, anticipate what each of the following lines will do, and then record what happens. Score yourself, giving yourself one point for each one you anticipate correctly:

```
>>> import turtle
>>> wn = turtle.Screen()
>>> tess = turtle.Turtle()
>>> tess.right(90)
>>> tess.left(3600)
>>> tess.right(-90)
>>> tess.speed(10)
>>> tess.left(3600)
>>> tess.speed(0)
>>> tess.left(3645)
>>> tess.forward(-100)
```

11. Write a program to draw a shape like this:

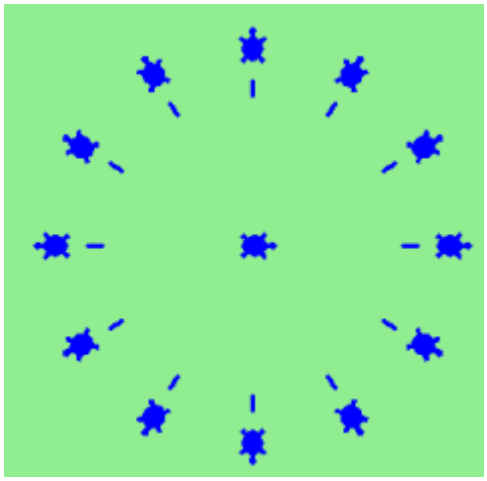
Star

Hints:

- Try this on a piece of paper, moving and turning your cellphone as if it was a turtle. Watch how many complete rotations your cellphone makes before you complete the star. Since each full rotation is 360 degrees, you can figure out the total number of degrees that your phone was rotated through. If you divide that by 5, because there are five points to the star, you'll know how many degrees to turn the turtle at each point.

- You can hide a turtle behind its invisibility cloak if you don't want it shown. It will still draw its lines if its pen is down. The method is invoked as `tess.hideturtle()`. To make the turtle visible again, use `tess.showturtle()`.

12. Write a program to draw a face of a clock that looks something like this:



Clock face

13. Create a turtle, and assign it to a variable. When you ask for its type, what do you get?
14. What is the collective noun for turtles? (Hint: they don't come in *herds*.)
15. What the collective noun for pythons? Is a python a viper? Is a python venomous?

Chapter 4: Functions

4.1. Functions

In Python, a **function** is a named sequence of statements that belong together. Their primary purpose is to help us organize programs into chunks that match how we think about the problem.

The syntax for a **function definition** is:

```
def NAME( PARAMETERS ):
    STATEMENTS
```

We can make up any names we want for the functions we create, except that we can't use a name that is a Python keyword, and the names must follow the rules for legal identifiers.

There can be any number of statements inside the function, but they have to be indented from the `def`. In the examples in this book, we will use the standard indentation of four spaces. Function definitions are the second of several **compound statements** we will see, all of which have the same pattern:

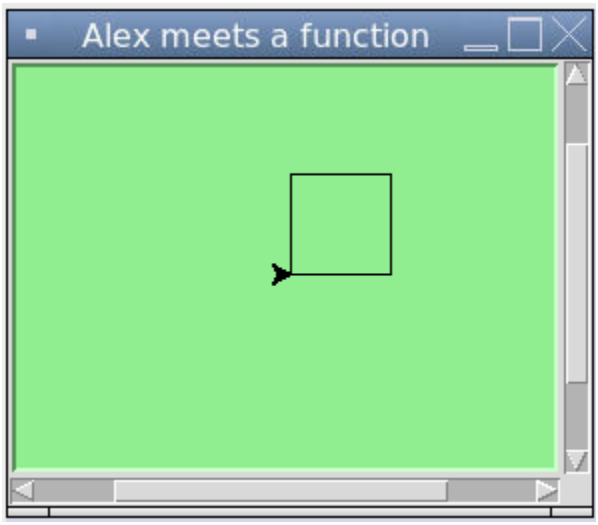
1. A header line which begins with a keyword and ends with a colon.
2. A **body** consisting of one or more Python statements, each indented the same amount — the *Python style guide recommends 4 spaces* — from the header line.

We've already seen the `for` loop which follows this pattern.

So looking again at the function definition, the keyword in the header is `def`, which is followed by the name of the function and some *parameters* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters separated from one another by commas. In either case, the parentheses are required. The parameters specifies what information, if any, we have to provide in order to use the new function.

Suppose we're working with turtles, and a common operation we need is to draw squares. "Draw a square" is an abstraction, or a mental chunk, of a number of smaller steps. So let's write a function to capture the pattern of this "building block":

```
import turtle

def draw_square(t, sz):
    """Make turtle t draw a square of sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)


wn = turtle.Screen()        # Set up the window and its attributes
wn.bgcolor("lightgreen")
wn.title("Alex meets a function")

alex = turtle.Turtle()      # Create alex
draw_square(alex, 50)       # Call the function to draw the square
wn.mainloop()
```



alex function

This function is named `draw_square`. It has two parameters: one to tell the function which turtle to move around, and the other to tell it the size of the square we want drawn. Make sure you know where the body of the function ends — it depends on the indentation, and the blank lines don't count for this purpose!

Docstrings for documentation

If the first thing after the function header is a string, it is treated as a **docstring** and gets special treatment in Python and in some programming tools. For example, when we type a built-in function name with an unclosed parenthesis in Repl.it, a tooltip pops up, telling us what arguments the function takes, and it shows us any other text contained in the docstring.

Docstrings are the key way to document our functions in Python and the documentation part is important. Because whoever calls our function shouldn't have to need to know what is going on in the function or how it works; they just need to know what arguments our function takes, what it does, and what the expected result is. Enough to be able to use the function without having to look underneath. This goes back to the concept of abstraction of which we'll talk more about.

Docstrings are usually formed using triple-quoted strings as they allow us to easily expand the docstring later on should we want to write more than a one-liner.

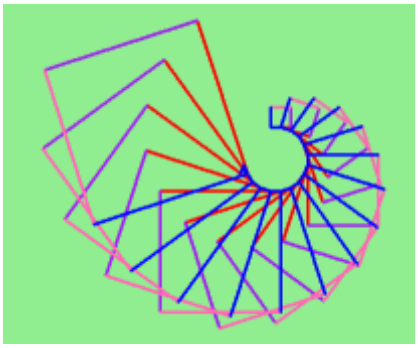
Just to differentiate from comments, a string at the start of a function (a docstring) is retrievable by Python tools at runtime. By contrast, comments are completely eliminated when the program is parsed.

Defining a new function does not make the function run. To do that we need a **function call**. We've already seen how to call some built-in functions like **`print`**, **`range`** and **`int`**. Function calls contain the name of the function being executed followed by a list of values, called *arguments*, which are assigned to the parameters in the function definition. So in the second last line of the program, we call the function, and pass alex as the turtle to be manipulated, and `50` as the size of the square we want. While the function is executing, then, the variable `sz` refers to the value `50`, and the variable t refers to the same turtle instance that the variable `alex` refers to.

Once we've defined a function, we can call it as often as we like, and its statements will be executed each time we call it. And we could use it to get any of our turtles to draw a square. In the next example, we've changed the draw_square function a little, and we get tess to draw 15 squares, with some variations.

```
1  import turtle
2
3  def draw_multicolor_square(t, sz):
4      """Make turtle t draw a multi-color square of sz."""
5      for i in ["red", "purple", "hotpink", "blue"]:
6          t.color(i)
7          t.forward(sz)
8          t.left(90)
9
10 wn = turtle.Screen()        # Set up the window and its attributes
11 wn.bgcolor("lightgreen")
12
13 tess = turtle.Turtle()      # Create tess and set some attributes
14 tess.pensize(3)
15
16 size = 20                   # Size of the smallest square
17 for i in range(15):
18     draw_multicolor_square(tess, size)
19     size = size + 10        # Increase the size for next time
```

```
20      tess.forward(10)        # Move tess along a little
21      tess.right(18)          #    and give her some turn
22
23  wn.mainloop()
```



Draw multicolor square

4.2. Functions can call other functions

Let's assume now we want a function to draw a rectangle. We need to be able to call the function with different arguments for width and height. And, unlike the case of the square, we cannot repeat the same thing 4 times, because the four sides are not equal.

So we eventually come up with this rather nice code that can draw a rectangle.

```
1  def draw_rectangle(t, w, h):
2      """Get turtle t to draw a rectangle of width w and height h."""
3      for i in range(2):
4          t.forward(w)
5          t.left(90)
6          t.forward(h)
7          t.left(90)
```

The parameter names are deliberately chosen as single letters to ensure they're not misunderstood. In real programs, once we've had more experience, we will insist on better variable names than this. But the point is that the program doesn't "understand" that we're drawing a rectangle, or that the parameters represent the width and the height. Concepts like rectangle, width, and height are the meaning we humans have, not concepts that the program or the computer understands.

Thinking like a scientist involves looking for patterns and relationships. In the code above, we've done that to some extent. We did not just draw four sides. Instead, we spotted that we could draw the rectangle as two halves, and used a loop to repeat that pattern twice.

But now we might spot that a square is a special kind of rectangle. We already have a function that draws a rectangle, so we can use that to draw our square.

```
def draw_square(tx, sz):        # A new version of draw_square
    draw_rectangle(tx, sz, sz)
```

There are some points worth noting here:

- Functions can call other functions.
- Rewriting `draw_square` like this captures the relationship that we've spotted between squares and rectangles.
- A caller of this function might say `draw_square(tess, 50)`. The parameters of this function, `tx` and `sz`, are assigned the values of the tess object, and the `int 50` respectively.
- In the body of the function they are just like any other variable.
- When the call is made to `draw_rectangle`, the values in variables `tx` and `sz` are fetched first, then the call happens. So as we enter the top of function `draw_rectangle`, its variable `t` is assigned the `tess` object, and `w` and `h` in that function are both given the value `50`.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives us an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command. The function (including its name) can capture our mental chunking, or *abstraction*, of the problem.
2. Creating a new function can make a program smaller by eliminating repetitive code.

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

4.3. Flow of execution

In order to ensure that a function is defined before its first use, we have to know the order in which statements are executed, which is called the **flow of execution**. We've already talked about this a little in the previous chapter.

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. Although it is not common, we can define one function inside another. In this case, the inner definition isn't executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until we 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 adept 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.

What's the moral of this sordid tale? When we read a program, don't read from top to bottom. Instead, follow the flow of execution.

Watch the flow of execution in action

Repl.it does not have "single-stepping" functionality. For this we would recommend a different IDE like PyScripter[6].

In PyScripter, we can watch the flow of execution by "single-stepping" through any program. PyScripter will highlight each line of code just before it is about to be executed.

PyScripter also lets us hover the mouse over any variable in the program, and it will pop up the current value of that variable. So this makes it easy to inspect the "state snapshot" of the program — the current values that are assigned to the program's variables.

This is a powerful mechanism for building a deep and thorough understanding of what is happening at each step of the way. Learn to use the single-stepping feature well, and be mentally proactive: as you work through the code, challenge yourself before each step: *"What changes will this line make to any variables in the program?"* and "Where will flow of execution go next?"

Let us go back and see how this works with the program above that draws 15 multicolor squares. First, we're going to add one line of magic below the import statement — not strictly necessary, but it will make our lives much simpler, because it prevents stepping into the module containing the turtle code.

```
1 import turtle
2 __import__("turtle").__traceable__ = False
```

Now we're ready to begin. Put the mouse cursor on the line of the program where we create the turtle screen, and press the F4 key. This will run the Python program up to, but not including, the line where we have the cursor. Our program will "break" now, and provide a highlight on the next line to be executed, something like this:

[6] https://sourceforge.net/projects/pyscripter/

```
 1 import turtle
 2 __import__("turtle").__traceable__ = False
 3
 4 def draw_multicolor_square(t, sz):
 5     """Make turtle t draw a multi-color square of sz."""
 6     for i in ["red", "purple", "hotpink", "blue"]:
 7         t.color(i)
 8         t.forward(sz)
 9         t.left(90)
10
11 wn = turtle.Screen()        # Set up the window and its attributes
12 wn.bgcolor("lightgreen")
13
14 tess = turtle.Turtle()      # Create tess and set some attributes
15 tess.pensize(3)
16
17 size = 20                   # Size of the smallest square
18 for i in range(15):
19     draw_multicolor_square(tess, size)
20     size = size + 10        # Increase the size for next time
21     tess.forward(10)        # Move tess along a little
22     tess.right(18)          #  ...  and give her some extra turn
23
24 wn.mainloop()
25
```

PyScripter Breakpoint

At this point we can press the F7 key (*step into*) repeatedly to single step through the code. Observe as we execute lines 10, 11, 12, … how the turtle window gets created, how its canvas color is changed, how the title gets changed, how the turtle is created on the canvas, and then how the flow of execution gets into the loop, and from there into the function, and into the function's loop, and then repeatedly through the body of that loop.

While we do this, we can also hover our mouse over some of the variables in the program, and confirm that their values match our conceptual model of what is happening.

After a few loops, when we're about to execute line 20 and we're starting to get bored, we can use the key F8 to "step over" the function we are calling. This executes all the statements in the function, but without having to step through each one. We always have the choice to either "go for the detail", or to "take the high-level view" and execute the function as a single chunk.

There are some other options, including one that allow us to resume execution without further stepping. Find them under the *Run* menu of PyScripter.

4.4. Functions that require arguments

Most functions require arguments: the arguments provide for generalization. For example, if we want to find the absolute value of a number, we have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
>>> abs(5)
5
>>> abs(-5)
5
```

In this example, the arguments to the `abs` function are `5` and `-5`.

Some functions take more than one argument. For example the built-in function `pow` takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

Another built-in function that takes more than one argument is `max`.

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

`max` can be passed any number of arguments, separated by commas, and will return the largest value passed. The arguments can be either simple values or expressions. In the last example, `503` is returned, since it is larger than `33`, `125`, and `1`.

4.5. Functions that return values

All the functions in the previous section return values. Furthermore, functions like `range`, `int`, `abs` all return values that can be used to build more complex expressions.

So an important difference between these functions and one like `draw_square` is that `draw_square` was not executed because we wanted it to compute a value — on the contrary, we wrote `draw_square` because we wanted it to execute a sequence of steps that caused the turtle to draw.

A function that returns a value is called a **fruitful function** in this book. The opposite of a fruitful function is **void function** — one that is not executed for its resulting value, but is executed because it does something useful. (Languages like Java, C#, C and C++ use the term "void function", other languages like Pascal call it a **procedure**.) Even though void functions are not executed for their resulting value, Python always wants to return something. So if the programmer doesn't arrange to return a value, Python will automatically return the value `None`.

How do we write our own fruitful function? In the exercises at the end of chapter 2 we saw the standard formula for compound interest, which we'll now write as a fruitful function:

$$P' = P\left(1 + \frac{r}{n}\right)^{nt}$$

where:

P is the original principal sum

P′ is the new principal sum

r is the nominal annual interest rate

n is the compounding frequency

t is the overall length of time the interest is applied (expressed using the same time units as *r*, usually years).

Compound interest

```
def final_amt(p, r, n, t):
    """
      Apply the compound interest formula to p
       to produce the final amount.
    """

    a = p * (1 + r/n) ** (n*t)
    return a          # This is new, and makes the function fruitful.

# now that we have the function above, let us call it.
toInvest = float(input("How much do you want to invest?"))
fnl = final_amt(toInvest, 0.08, 12, 5)
print("At the end of the period you'll have", fnl)
```

- The return statement is followed by an expression (`a` in this case). This expression will be evaluated and returned to the caller as the "fruit" of calling this function.
- We prompted the user for the principal amount. The type of `toInvest` is a string, but we need a number before we can work with it. Because it is money, and could have decimal places, we've used the `float` type converter function to parse the string and return a float.
- Notice how we entered the arguments for 8% interest, compounded 12 times per year, for 5 years.
- When we run this, we get the output

```
1    At the end of the period you'll have 14898.457083
```

This is a bit messy with all these decimal places, but remember that Python doesn't understand that we're working with money: it just does the calculation to the best of its ability, without rounding. Later we'll see how to format the string that is printed in such a way that it does get nicely rounded to two decimal places before printing.

- The line `toInvest = float(input("How much do you want to invest?"))` also shows yet another example of *composition* — we can call a function like `float`, and its arguments can be the results of other function calls (like input) that we've called along the way.

Notice something else very important here. The name of the variable we pass as an argument — `toInvest` — has nothing to do with the name of the parameter — `p`. It is as if `p = toInvest` is executed when `final_amt` is called. It doesn't matter what the value was named in the caller, in `final_amt` its name is `p`.

These short variable names are getting quite tricky, so perhaps we'd prefer one of these versions instead:

```
1  def final_amt_v2(principalAmount, nominalPercentageRate,
2                                    numTimesPerYear, years):
3      a = principalAmount * (1 + nominalPercentageRate /
4                           numTimesPerYear) ** (numTimesPerYear*years)
5      return a
6
7  def final_amt_v3(amt, rate, compounded, years):
8      a = amt * (1 + rate/compounded) ** (compounded*years)
9      return a
```

They all do the same thing. Use your judgement to write code that can be best understood by other humans! Short variable names are more economical and sometimes make code easier to read: `E = mc2` would not be nearly so memorable if Einstein had used longer variable names! If you do prefer short names, make sure you also have some comments to enlighten the reader about what the variables are used for.

4.6. Variables and parameters are local

When we create a **local variable** inside a function, it only exists inside the function, and we cannot use it outside. For example, consider again this function:

```
def final_amt(p, r, n, t):
    a = p * (1 + r/n) ** (n*t)
    return a
```

If we try to use `a`, outside the function, we'll get an error:

```
>>> a
NameError: name 'a' is not defined
```

The variable `a` is local to `final_amt`, and is not visible outside the function.

Additionally, `a` only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates, the local variables are destroyed.

Parameters are also local, and act like local variables. For example, the lifetimes of `p`, `r`, `n`, `t` begin when `final_amt` is called, and the lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

4.7. Turtles Revisited

Now that we have fruitful functions, we can focus our attention on reorganizing our code so that it fits more nicely into our mental chunks. This process of rearrangement is called **refactoring** the code.

Two things we're always going to want to do when working with turtles is to create the window for the turtle, and to create one or more turtles. We could write some functions to make these tasks easier in future:

```
def make_window(colr, ttle):
    """
      Set up the window with the given background color and title.
      Returns the new window.
    """
    w = turtle.Screen()
    w.bgcolor(colr)
    w.title(ttle)
    return w


def make_turtle(colr, sz):
```

```
    """
      Set up a turtle with the given color and pensize.
      Returns the new turtle.
    """
    t = turtle.Turtle()
    t.color(colr)
    t.pensize(sz)
    return t


wn = make_window("lightgreen", "Tess and Alex dancing")
tess = make_turtle("hotpink", 5)
alex = make_turtle("black", 1)
dave = make_turtle("yellow", 2)
```

The trick about refactoring code is to anticipate which things we are likely to want to change each time we call the function: these should become the parameters, or changeable parts, of the functions we write.

4.8. Glossary

argument

A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function. The argument can be the result of an expression which may involve operators, operands and calls to other fruitful functions.

body

The second part of a compound statement. The body consists of a sequence of statements all indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

compound statement

A statement that consists of two parts:

1. header - which begins with a keyword determining the statement type, and ends with a colon.
2. body - containing one or more statements indented the same amount from the header.

The syntax of a compound statement looks like this:

```
keyword ... :
    statement
    statement ...
```

docstring

A special string that is attached to a function as its `__doc__` attribute. Tools like Repl.it can use docstrings to provide documentation or hints for the programmer. When we get to modules, classes, and methods, we'll see that docstrings can also be used there.

flow of execution

The order in which statements are executed during a program run.

frame

A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

function

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

function call

A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

function composition

Using the output from one function call as the input to another.

function definition

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

fruitful function

A function that returns a value when it is called.

header line

The first part of a compound statement. A header line begins with a keyword and ends with a colon (:)

import statement

A statement which permits functions and variables defined in another Python module to be brought into the environment of another script. To use the features of the turtle, we need to first import the turtle module.

lifetime

Variables and objects have lifetimes — they are created at some point during program execution, and will be destroyed at some time.

local variable

A variable defined inside a function. A local variable can only be used inside its function. Parameters of a function are also a special kind of local variable.

parameter

A name used inside a function to refer to the value which was passed to it as an argument.

refactor

A fancy word to describe reorganizing our program code, usually to make it more understandable. Typically, we have a program that is already working, then we go back to "tidy it up". It often involves choosing better variable names, or spotting repeated patterns and moving that code into a function.

stack diagram

A graphical representation of a stack of functions, their variables, and the values to which they refer.

traceback

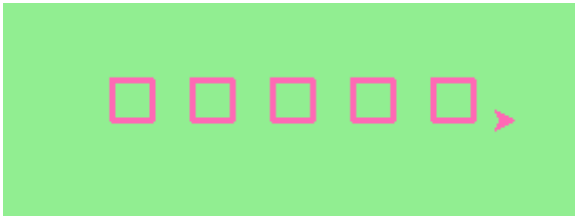
A list of the functions that are executing, printed when a runtime error occurs. A traceback is also commonly referred to as a *stack trace*, since it lists the functions in the order in which they are stored in the runtime stack.[7]

void function

The opposite of a fruitful function: one that does not return a value. It is executed for the work it does, rather than for the value it returns.

4.9. Exercises

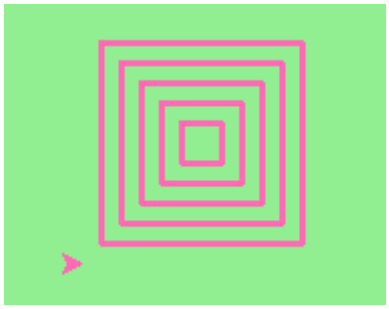
1. Write a void (non-fruitful) function to draw a square. Use it in a program to draw the image shown below. Assume each side is 20 units. (*Hint: notice that the turtle has already moved away from the ending point of the last square when the program ends.*)



Five Squares

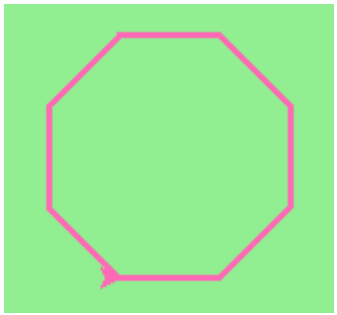
[7] http://en.wikipedia.org/wiki/Runtime_stack

2. Write a program to draw this. Assume the innermost square is 20 units per side, and each successive square is 20 units bigger, per side, than the one inside it.



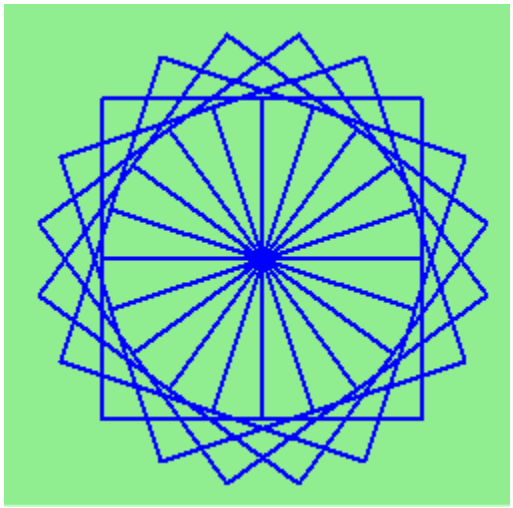
Nested Squares

3. Write a void function `draw_poly(t, n, sz)` which makes a turtle draw a regular polygon. When called with `draw_poly(tess, 8, 50)`, it will draw a shape like this:



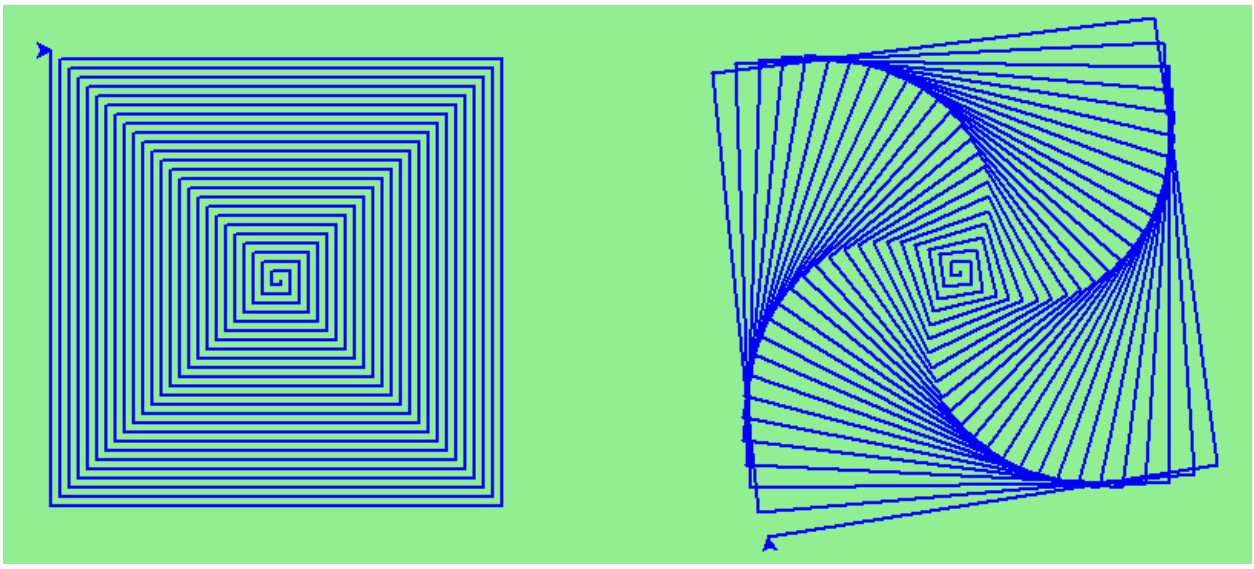
Regular polygon

4. Draw this pretty pattern.



Regular Polygon

5. The two spirals in this picture differ only by the turn angle. Draw both.

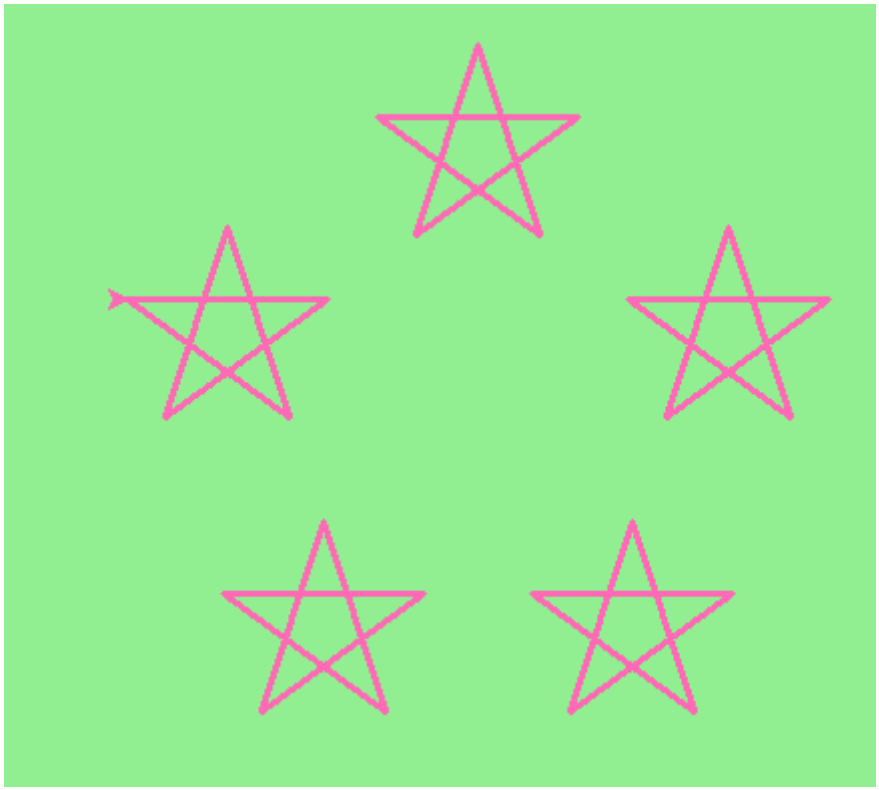


Spirals

6. Write a void function `draw_equitriangle(t, sz)` which calls `draw_poly` from the previous question to have its turtle draw a equilateral triangle.
7. Write a fruitful function `sum_to(n)` that returns the sum of all integer numbers up to and including `n`. So `sum_to(10)` would be `1+2+3...+10` which would return the value `55`.
8. Write a function `area_of_circle(r)` which returns the area of a circle of radius `r`.
9. Write a void function to draw a star, where the length of each side is 100 units. (*Hint: You should turn the turtle by 144 degrees at each point.*)

Star

10. Extend your program above. Draw five stars, but between each, pick up the pen, move forward by 350 units, turn right by 144, put the pen down, and draw the next star. You'll get something like this:



Five Stars

What would it look like if you didn't pick up the pen?