FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Contents

Chapter 1

Why should you learn to write programs?

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

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

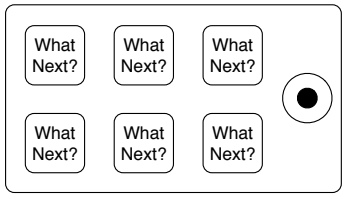

Figure 1.1: Personal Digital Assistant

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

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

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

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

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

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

1.1 Creativity and motivation

While this book is not intended for professional programmers, professional programming can be a very rewarding job both financially and personally. Building useful, elegant, and clever programs for others to use is a very creative activity. Your computer or Personal Digital Assistant (PDA) usually contains many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process. In some situations, when you choose a piece of software, the programmers are directly compensated because of your choice.

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

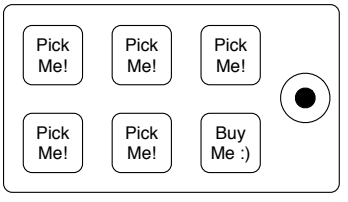

Figure 1.2: Programmers Talking to You

For now, our primary motivation is not to make money or please end users, but instead for us to be more productive in handling the data and information that we will encounter in our lives. When you first start, you will be both the programmer and the end user of your programs. As you gain skill as a programmer and programming feels more creative to you, your thoughts may turn toward developing programs for others.

1.2 Computer hardware architecture

Before we start learning the language we speak to give instructions to computers to develop software, we need to learn a small amount about how computers are built. If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:

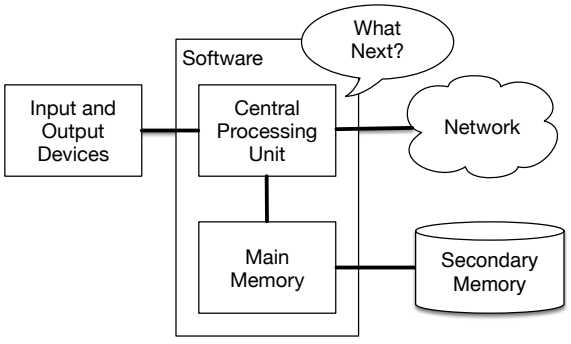

Figure 1.3: Hardware Architecture

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

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

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

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

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

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

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

As a programmer, your job is to use and orchestrate each of these resources to solve the problem that you need to solve and analyze the data you get from the solution. As a programmer you will mostly be "talking" to the CPU and telling it what to do next. Sometimes you will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.

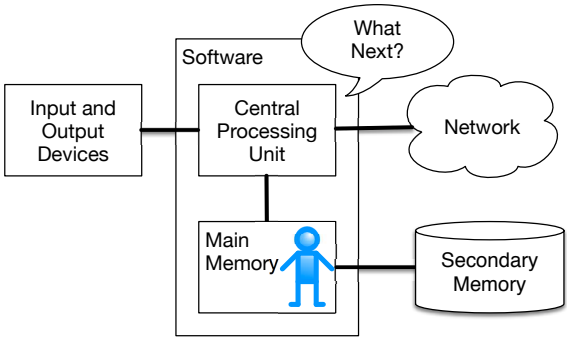

Figure 1.4: Where Are You?

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

1.3 Understanding programming

In the rest of this book, we will try to turn you into a person who is skilled in the art of programming. In the end you will be a *programmer* - perhaps not a professional programmer, but at least you will have the skills to look at a data/information analysis problem and develop a program to solve the problem.

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

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

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++. The

new programming language has very different vocabulary and grammar but the problem-solving skills will be the same across all programming languages.

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

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

1.4 Words and sentences

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

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

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

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

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

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

[1] http://xkcd.com/231/

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

And we have even written our first syntactically correct Python sentence. Our sentence starts with the function *print* followed by a string of text of our choosing enclosed in single quotes. The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

1.5 Conversing with Python

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

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

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

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

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

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

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

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

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

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
File "<stdin>", line 1
  print 'We will have a feast tonight unless you say
                                                    ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

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

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

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

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

```
>>> good-bye
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
  if you don't mind, I need to leave
           ^
SyntaxError: invalid syntax
>>> quit()
```

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

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

1.6 Terminology: Interpreter and compiler

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

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

```
00101000111010010010101000000001111
11100110000011101010010101101101
...
```

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

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

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

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

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

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

```
>>> print(y)
42
>>>
```

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

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

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

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

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

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

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

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

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

```
C:\Python35\python.exe
```

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

1.7 Writing a program

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

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

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

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

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

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

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

1.8 What is a program?

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

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

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

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

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

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

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

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

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

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

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

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

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

1.9 The building blocks of programs

In the next few chapters, we will learn more about the vocabulary, sentence structure, paragraph structure, and story structure of Python. We will learn about the

powerful capabilities of Python and how to compose those capabilities together to create useful programs.

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

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

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

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

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

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

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

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

The word counting program above directly uses all of these patterns except for one.

1.10 What could possibly go wrong?

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

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

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

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

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

Python is much more like a dog, loving you unconditionally, having a few key words that it understands, looking you with a sweet look on its face (>>>), and waiting for you to say something it understands. When Python says "SyntaxError: invalid syntax", it is simply wagging its tail and saying, "You seemed to say something but I just don't understand what you meant, but please keep talking to me (>>>)."

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

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

Logic errors A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another. A good example of a logic error might be, "take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle."

Semantic errors A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you *intended* for it to do. A simple example would be if you were giving a person directions to a restaurant and said, "... when you reach the intersection with the gas station, turn left and go one mile and the restaurant is a red building on your left." Your friend is very late and calls you to tell you that they are on a farm and walking around behind a barn, with no sign of a restaurant. Then you say "did you turn left or right at the gas station?" and they say, "I followed your directions perfectly, I have them written down, it says turn left and go one mile at the gas station." Then you say, "I am very sorry, because while my instructions were syntactically correct, they sadly contained a small but undetected semantic error.".

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

1.11 Debugging

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

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

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

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

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

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

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

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

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

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

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best

option is to retreat, simplifying the program until you get to something that works and that you understand.

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

1.12 The learning journey

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

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

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

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

1.13 Glossary

bug An error in a program.

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

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

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

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

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

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

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

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

parse To examine a program and analyze the syntactic structure.

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

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

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

program A set of instructions that specifies a computation.

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

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

semantics The meaning of a program.

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

source code A program in a high-level language.

1.14 Exercises

Exercise 1: What is the function of the secondary memory in a computer?

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

Exercise 2: What is a program?

Exercise 3: What is the difference between a compiler and an interpreter?

Exercise 4: Which of the following contains "machine code"?

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

Exercise 5: What is wrong with the following code:

```
>>> primt 'Hello world!'
File "<stdin>", line 1
  primt 'Hello world!'
                      ^
SyntaxError: invalid syntax
>>>
```

Exercise 6: Where in the computer is a variable such as "x" stored after the following Python line finishes?

```
x = 123
```

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

Exercise 7: What will the following program print out:

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

a) 43
b) 44
c) x + 1
d) Error because x = x + 1 is not possible mathematically

Exercise 8: Explain each of the following using an example of a human capability: (1) Central processing unit, (2) Main Memory, (3) Secondary Memory, (4) Input Device, and (5) Output Device. For example, "What is the human equivalent to a Central Processing Unit"?

Exercise 9: How do you fix a "Syntax Error"?

Chapter 2

Variables, expressions, and statements

2.1 Values and types

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

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

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

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

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

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

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

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

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

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

They're strings.

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

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

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

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

2.2 Variables

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

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

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

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

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

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

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

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

2.3 Variable names and keywords

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

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

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

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

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

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

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

Python reserves 35 keywords:

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

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

2.4 Statements

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

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

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

For example, the script

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

produces the output

```
1
2
```

The assignment statement produces no output.

2.5 Operators and operands

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

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

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

There has been a change in the division operator between Python 2.x and Python 3.x. In Python 3.x, the result of this division is a floating point result:

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

The division operator in Python 2.0 would divide two integers and truncate the result to an integer:

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

To obtain the same answer in Python 3.0 use floored (`//` integer) division.

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

In Python 3.0 integer division functions much more as you would expect if you entered the expression on a calculator.

2.6 Expressions

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

```
17
x
x + 17
```

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

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

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

Exercise 1: Type the following statements in the Python interpreter to see what they do:

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

2.7 Order of operations

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

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

- *E*xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.
- *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So the expression `5-3-1` is 1, not 3, because the `5-3` happens first and then `1` is subtracted from 2.

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

2.8 Modulus operator

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

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

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

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

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

2.9 String operations

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

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

```
>>> second = '150'
>>> print(first + second)
100150
```

The * operator also works with strings by multiplying the content of a string by an integer. For example:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

2.10 Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `input` that gets input from the keyboard[1]. When this function is called, the program stops and waits for the user to type something. When the user presses `Return` or `Enter`, the program resumes and `input` returns what the user typed as a string.

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

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

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

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

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

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

[1] In Python 2.0, this function was named `raw_input`.

```
>>> int(speed)
17
>>> int(speed) + 5
22
```

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

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

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

2.11 Comments

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

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

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

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

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

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

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

This comment is redundant with the code and useless:

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

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

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

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

2.12 Choosing mnemonic variable names

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

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

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

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

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

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[2] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

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

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

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

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

The following code is equivalent to the above code:

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

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

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

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

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

2.13 Debugging

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

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

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

>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

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

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

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

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

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

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

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

2.14 Glossary

assignment A statement that assigns a value to a variable.

concatenate To join two operands end to end.

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

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

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

floating point A type that represents numbers with fractional parts.

integer A type that represents whole numbers.

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

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

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

operand One of the values on which an operator operates.

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

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

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

string A type that represents sequences of characters.
type A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).
value One of the basic units of data, like a number or string, that a program manipulates.
variable A name that refers to a value.

2.15 Exercises

Exercise 2: Write a program that uses `input` to prompt a user for their name and then welcomes them.

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

Exercise 3: Write a program to prompt the user for hours and rate per hour to compute gross pay.

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

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

Exercise 4: Assume that we execute the following assignment statements:

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

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

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

Use the Python interpreter to check your answers.

Exercise 5: Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.

Chapter 3

Conditional execution

3.1 Boolean expressions

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

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

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

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

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

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

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

3.2 Logical operators

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

`x > 0 and x < 10`

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

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

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

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

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

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

3.3 Conditional execution

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

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

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

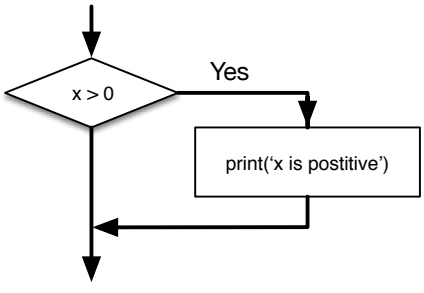

Figure 3.1: If Logic

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

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

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

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

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

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

When using the Python interpreter, you must leave a blank line at the end of a block, otherwise Python will return an error:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
... print('Done')
  File "<stdin>", line 3
    print('Done')
        ^
SyntaxError: invalid syntax
```

A blank line at the end of a block of statements is not necessary when writing and executing a script, but it may improve readability of your code.

3.4 Alternative execution

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

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

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

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

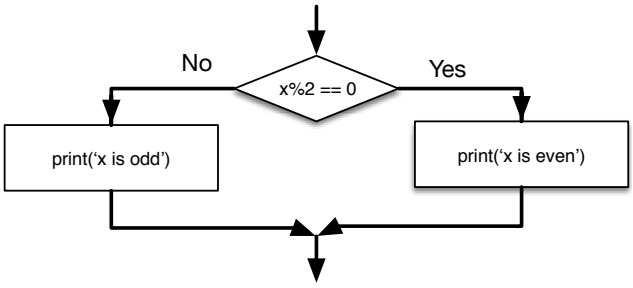

Figure 3.2: If-Then-Else Logic

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

3.5 Chained conditionals

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

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

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

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

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

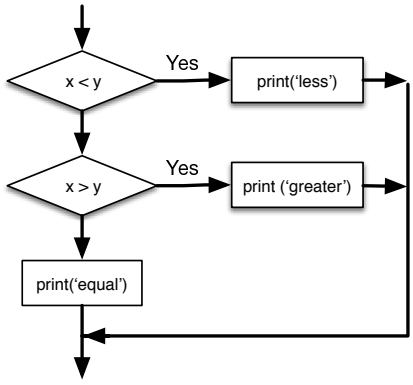

Figure 3.3: If-Then-ElseIf Logic

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

3.6 Nested conditionals

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

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

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

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

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

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

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

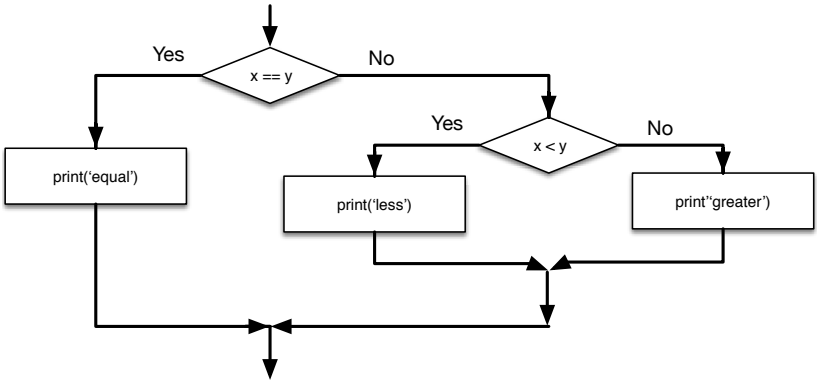

Figure 3.4: Nested If Statements

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

3.7 Catching exceptions using try and except

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

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

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

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

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

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

# Code: http://www.py4e.com/code3/fahren.py
```

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

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

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

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

We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py
```

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

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a `try` statement is called *catching* an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short-circuit evaluation of logical expressions

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

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

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

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

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

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

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

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

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

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

3.9 Debugging

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

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

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

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

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

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

3.10 Glossary

body The sequence of statements within a compound statement.
boolean expression An expression whose value is either `True` or `False`.
branch One of the alternative sequences of statements in a conditional statement.

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

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

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

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

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

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

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

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

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

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

3.11 Exercises

Exercise 1: Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

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

Exercise 2: Rewrite your pay program using `try` and `except` so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

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

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

Exercise 3: Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
 Score   Grade
```

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

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

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

Chapter 4

Functions

4.1 Function calls

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

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

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

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

4.2 Built-in functions

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

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

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

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

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

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

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

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

4.3 Type conversion functions

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

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

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

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

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

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

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

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

4.4 Math functions

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

```
>>> import math
```

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

```
>>> print(math)
<module 'math' (built-in)>
```

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

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

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

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

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

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

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

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

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5 Random numbers

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

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

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

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

```
import random

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

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

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Exercise 1: Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

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

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

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

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

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

4.6 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

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

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

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

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

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

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

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

Defining a function creates a variable with the same name.

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

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

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

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

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

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

And then call `repeat_lyrics`:

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

But that's not really how the song goes.

4.7 Definitions and uses

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

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


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py
```

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

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

Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 3: Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

4.8 Flow of execution

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

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

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

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

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

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

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

4.9 Parameters and arguments

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

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

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

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

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

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

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

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

You can also use a variable as an argument:

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

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

4.10 Fruitful functions and void functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Code: http://www.py4e.com/code3/addtwo.py
```

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

4.11 Why functions?

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

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

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

4.12 Debugging

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

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

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

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

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

4.13 Glossary

algorithm A general process for solving a category of problems.

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

body The sequence of statements inside a function definition.

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

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

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

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

fruitful function A function that returns a value.

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

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

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

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

header The first line of a function definition.

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

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

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

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

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

void function A function that does not return a value.

4.14 Exercises

Exercise 4: What is the purpose of the "def" keyword in Python?

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

Exercise 5: What will the following Python program print out?

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

def jane():
   print("ABC")

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

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

Exercise 6: Rewrite your pay computation with time-and-a-half for overtime and create a function called `computepay` which takes two parameters (`hours` and `rate`).

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

Exercise 7: Rewrite the grade program from the previous chapter using a function called `computegrade` that takes a score as its parameter and returns a grade as a string.

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

Enter score: 0.95
A

Enter score: perfect
Bad score
```

```
Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

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