

# 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](https://bit.ly/VTU_TELEGRAM)

Contact: MAIL: [futurevisionbie@gmail.com](mailto:futurevisionbie@gmail.com)

INSTAGRAM: [www.instagram.com/hemanthraj\\_hemu/](http://www.instagram.com/hemanthraj_hemu/)

INSTAGRAM: [www.instagram.com/futurevisionbie/](http://www.instagram.com/futurevisionbie/)

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

# Contents

<b>Preface</b>	<b>v</b>
<b>1 The way of the program</b>	<b>1</b>
1.1 What is a program? . . . . .	1
1.2 Running Python . . . . .	2
1.3 The first program . . . . .	3
1.4 Arithmetic operators . . . . .	3
1.5 Values and types . . . . .	4
1.6 Formal and natural languages . . . . .	4
1.7 Debugging . . . . .	6
1.8 Glossary . . . . .	6
1.9 Exercises . . . . .	7
<b>2 Variables, expressions and statements</b>	<b>9</b>
2.1 Assignment statements . . . . .	9
2.2 Variable names . . . . .	9
2.3 Expressions and statements . . . . .	10
2.4 Script mode . . . . .	11
2.5 Order of operations . . . . .	11
2.6 String operations . . . . .	12
2.7 Comments . . . . .	12
2.8 Debugging . . . . .	13
2.9 Glossary . . . . .	14
2.10 Exercises . . . . .	14

---

<b>3</b>	<b>Functions</b>	<b>17</b>
3.1	Function calls . . . . .	17
3.2	Math functions . . . . .	18
3.3	Composition . . . . .	19
3.4	Adding new functions . . . . .	19
3.5	Definitions and uses . . . . .	20
3.6	Flow of execution . . . . .	21
3.7	Parameters and arguments . . . . .	21
3.8	Variables and parameters are local . . . . .	22
3.9	Stack diagrams . . . . .	23
3.10	Fruitful functions and void functions . . . . .	24
3.11	Why functions? . . . . .	24
3.12	Debugging . . . . .	25
3.13	Glossary . . . . .	25
3.14	Exercises . . . . .	26
<b>4</b>	<b>Case study: interface design</b>	<b>29</b>
4.1	The turtle module . . . . .	29
4.2	Simple repetition . . . . .	30
4.3	Exercises . . . . .	31
4.4	Encapsulation . . . . .	32
4.5	Generalization . . . . .	32
4.6	Interface design . . . . .	33
4.7	Refactoring . . . . .	34
4.8	A development plan . . . . .	35
4.9	docstring . . . . .	35
4.10	Debugging . . . . .	36
4.11	Glossary . . . . .	36
4.12	Exercises . . . . .	37

<b>Contents</b>	<b>xv</b>
<b>5 Conditionals and recursion</b>	<b>39</b>
5.1 Floor division and modulus . . . . .	39
5.2 Boolean expressions . . . . .	40
5.3 Logical operators . . . . .	40
5.4 Conditional execution . . . . .	41
5.5 Alternative execution . . . . .	41
5.6 Chained conditionals . . . . .	41
5.7 Nested conditionals . . . . .	42
5.8 Recursion . . . . .	43
5.9 Stack diagrams for recursive functions . . . . .	44
5.10 Infinite recursion . . . . .	44
5.11 Keyboard input . . . . .	45
5.12 Debugging . . . . .	46
5.13 Glossary . . . . .	47
5.14 Exercises . . . . .	47
<b>6 Fruitful functions</b>	<b>51</b>
6.1 Return values . . . . .	51
6.2 Incremental development . . . . .	52
6.3 Composition . . . . .	54
6.4 Boolean functions . . . . .	54
6.5 More recursion . . . . .	55
6.6 Leap of faith . . . . .	57
6.7 One more example . . . . .	57
6.8 Checking types . . . . .	58
6.9 Debugging . . . . .	59
6.10 Glossary . . . . .	60
6.11 Exercises . . . . .	60

# Chapter 1

## The way of the program

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

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

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

### 1.1 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

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

**input:** Get data from the keyboard, a file, the network, or some other device.

**output:** Display data on the screen, save it in a file, send it over the network, etc.

**math:** Perform basic mathematical operations like addition and multiplication.

**conditional execution:** Check for certain conditions and run the appropriate code.

**repetition:** Perform some action repeatedly, usually with some variation.

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

## 1.2 Running Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I'll make suggestions for installing Python on your computer.

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started at <http://tinyurl.com/thinkpython2e>.

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

The Python **interpreter** is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `python` on a command line. When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 3.4.0 in this example, begins with 3, which indicates that you are running Python 3. If it begins with 2, you are running (you guessed it) Python 2.

The last line is a **prompt** that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

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

Now you're ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.

## 1.3 The first program

Traditionally, the first program you write in a new language is called “Hello, World!” because all it does is display the words “Hello, World!”. In Python, it looks like this:

```
>>> print('Hello, World!')
```

This is an example of a **print statement**, although it doesn’t actually print anything on paper. It displays a result on the screen. In this case, the result is the words

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don’t appear in the result.

The parentheses indicate that `print` is a function. We’ll get to functions in Chapter 3.

In Python 2, the `print` statement is slightly different; it is not a function, so it doesn’t use parentheses.

```
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that’s enough to get started.

## 1.4 Arithmetic operators

After “Hello, World”, the next step is arithmetic. Python provides **operators**, which are special symbols that represent computations like addition and multiplication.

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

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

The operator `/` performs division:

```
>>> 84 / 2
42.0
```

You might wonder why the result is `42.0` instead of `42`. I’ll explain in the next section.

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
>>> 6**2 + 6
42
```

In some other languages, `^` is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

```
>>> 6 ^ 2
4
```

I won’t cover bitwise operators in this book, but you can read about them at <http://wiki.python.org/moin/BitwiseOperators>.

## 1.5 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are 2, 42.0, and 'Hello, World!'.

These values belong to different **types**: 2 is an **integer**, 42.0 is a **floating-point number**, and 'Hello, World!' is a **string**, so-called because the letters it contains are strung together.

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

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word “class” is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type `int`, strings belong to `str` and floating-point numbers belong to `float`.

What about values like '2' and '42.0'? They look like numbers, but they are in quotation marks like strings.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

They're strings.

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

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

That's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

## 1.6 Formal and natural languages

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

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

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



Formal languages tend to have strict **syntax** rules that govern the structure of statements. For example, in mathematics the statement  $3 + 3 = 6$  has correct syntax, but  $3+ = 3\$6$  does not. In chemistry  $H_2O$  is a syntactically correct formula, but  $_2Zz$  is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with  $3+ = 3\$6$  is that \$ is not a legal token in mathematics (at least as far as I know). Similarly,  $_2Zz$  is not legal because there is no element with the abbreviation  $Zz$ .

The second type of syntax rule pertains to the way tokens are combined. The equation  $3 + /3$  is illegal because even though + and / are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engli\$h sentence with invalid t\*kens in it. This sentence all valid tokens has, but invalid structure with.

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

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

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

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

**literalness:** Natural languages are full of idiom and metaphor. If I say, "The penny dropped", there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

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

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

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

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## 1.7 Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

## 1.8 Glossary

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

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

**low-level language:** A programming language that is designed to be easy for a computer to run; also called “machine language” or “assembly language”.

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

**interpreter:** A program that reads another program and executes it

**prompt:** Characters displayed by the interpreter to indicate that it is ready to take input from the user.

**program:** A set of instructions that specifies a computation.

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

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

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

**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`).

**integer:** A type that represents whole numbers.

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

**string:** A type that represents sequences of characters.

**natural language:** Any one of the languages that people speak that evolved naturally.

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

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

**syntax:** The rules that govern the structure of a program.

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

**bug:** An error in a program.

**debugging:** The process of finding and correcting bugs.

## 1.9 Exercises

**Exercise 1.1.** *It is a good idea to read this book in front of a computer so you can try out the examples as you go.*

*Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?*

*This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.*

1. *In a print statement, what happens if you leave out one of the parentheses, or both?*
2. *If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?*
3. *You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?*

4. In math notation, leading zeros are ok, as in 09. What happens if you try this in Python? What about 011?
5. What happens if you have two values with no operator between them?

**Exercise 1.2.** Start the Python interpreter and use it as a calculator.

1. How many seconds are there in 42 minutes 42 seconds?
2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.
3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?

## Chapter 2

# Variables, expressions and statements

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.

### 2.1 Assignment statements

An **assignment statement** creates a new variable and gives it a value:

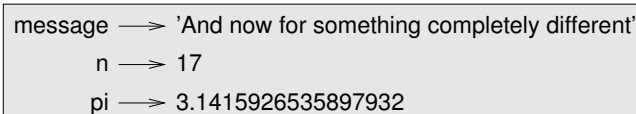
```
>>> message = 'And now for something completely different'  
>>> n = 17  
>>> pi = 3.1415926535897932
```

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

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2.1 shows the result of the previous example.

### 2.2 Variable names

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



```
message —> 'And now for something completely different'  
n —> 17  
pi —> 3.1415926535897932
```

Figure 2.1: State diagram.

Variable names can be as long as you like. They can contain both letters and numbers, but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.

The underscore character, `_`, can appear in a name. It is often used in names with multiple words, such as `your_name` or `airspeed_of_unladen_swallow`.

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 3 has these keywords:

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

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

## 2.3 Expressions and statements

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:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter **evaluates** it, which means that it finds the value of the expression. In this example, `n` has the value 17 and `n + 25` has the value 42.

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

The first line is an assignment statement that gives a value to `n`. The second line is a print statement that displays the value of `n`.

When you type a statement, the interpreter **executes** it, which means that it does whatever the statement says. In general, statements don't have values.

## 2.4 Script mode

So far we have run Python in **interactive mode**, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a **script** and then run the interpreter in **script mode** to execute the script. By convention, Python scripts have names that end with `.py`.

If you know how to create and run a script on your computer, you are ready to go. Otherwise I recommend using PythonAnywhere again. I have posted instructions for running in script mode at <http://tinyurl.com/thinkpython2e>.

Because Python provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Python as a calculator, you might type

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

The first line assigns a value to `miles`, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python evaluates the expression, but it doesn't display the result. To display the result, you need a `print` statement like this:

```
miles = 26.2
print(miles * 1.61)
```

This behavior can be confusing at first. To check your understanding, type the following statements in the Python interpreter and see what they do:

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

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

## 2.5 Order of operations

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

- Parentheses 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  $(\text{minute} * 100) / 60$ , even if it doesn't change the result.
- Exponentiation has the next highest precedence, so  $1 + 2**3$  is 9, not 27, and  $2 * 3**2$  is 18, not 36.
- Multiplication and Division have higher precedence than Addition and Subtraction. 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 (except exponentiation). So in the expression  $\text{degrees} / 2 * \text{pi}$ , the division happens first and the result is multiplied by pi. To divide by  $2\pi$ , you can use parentheses or write  $\text{degrees} / 2 / \text{pi}$ .

I don't work very hard to remember the precedence of operators. If I can't tell by looking at the expression, I use parentheses to make it obvious.

## 2.6 String operations

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
'chinese'-'food'      'eggs'/'easy'      'third'*'a charm'
```

But there are two exceptions, + and \*.

The + operator performs **string concatenation**, which means it joins the strings by linking them end-to-end. For example:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

The \* operator also works on strings; it performs repetition. For example, `'Spam'*3` is `'SpamSpamSpam'`. If one of the values is a string, the other has to be an integer.

This use of + and \* makes sense by analogy with addition and multiplication. Just as  $4*3$  is equivalent to  $4+4+4$ , we expect `'Spam'*3` to be the same as `'Spam'+'Spam'+'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

## 2.7 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 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 execution of 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 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 tradeoff.

## 2.8 Debugging

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

**Syntax error:** “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so (1 + 2) is legal, but 8) is a **syntax error**.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

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

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

**Semantic error:** The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

## 2.9 Glossary

**variable:** A name that refers to a value.

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

**state diagram:** A graphical representation of a set of variables and the values they refer to.

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

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

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

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

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

**execute:** To run a statement and do what it says.

**interactive mode:** A way of using the Python interpreter by typing code at the prompt.

**script mode:** A way of using the Python interpreter to read code from a script and run it.

**script:** A program stored in a file.

**order of operations:** Rules governing the order in which expressions involving multiple operators and operands are evaluated.

**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.

**syntax error:** An error in a program that makes it impossible to parse (and therefore impossible to interpret).

**exception:** An error that is detected while the program is running.

**semantics:** The meaning of a program.

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

## 2.10 Exercises

**Exercise 2.1.** Repeating my advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode and make errors on purpose to see what goes wrong.

- We've seen that `n = 42` is legal. What about `42 = n`?

- How about  $x = y = 1$ ?
- In some languages every statement ends with a semi-colon, `;`. What happens if you put a semi-colon at the end of a Python statement?
- What if you put a period at the end of a statement?
- In math notation you can multiply  $x$  and  $y$  like this:  $xy$ . What happens if you try that in Python?

**Exercise 2.2.** Practice using the Python interpreter as a calculator:

1. The volume of a sphere with radius  $r$  is  $\frac{4}{3}\pi r^3$ . What is the volume of a sphere with radius 5?
2. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?



# Chapter 3

## Functions

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.

### 3.1 Function calls

We have already seen one example of a **function call**:

```
>>> type(42)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this 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 also called the **return value**.

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

```
>>> int('32')
32
```

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

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

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

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

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

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

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

## 3.2 Math functions

Python has a `math` module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an **import statement**:

```
>>> import math
```

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

```
>>> 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 uses `math.log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The `math` module also provides `log`, which computes logarithms base  $e$ .

The second example finds the sine of radians. The variable name `radians` 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 180 and multiply by  $\pi$ :

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the `math` module. Its value is a floating-point approximation of  $\pi$ , accurate to about 15 digits.

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

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

## 3.3 Composition

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

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                 # wrong!
SyntaxError: can't assign to operator
```

## 3.4 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 run when the function is called.

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 underscore 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.

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, indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in double 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.

All quotation marks (single and double) must be "straight quotes", usually located next to Enter on the keyboard. "Curly quotes", like the ones in this sentence, are not legal in Python.

If you type a function definition in interactive mode, the interpreter prints dots (`. . .`) 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.

Defining a function creates a **function object**, which has type function:

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

### 3.5 Definitions and uses

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

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

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run 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 run it. In other words, the function definition has to run before the function gets called.

As an exercise, 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.

Now 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?

## 3.6 Flow of execution

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

Execution always begins at the first statement of the program. Statements are run 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 don't run 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, runs 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 run the statements in another function. Then, while running that new function, the program might have to run 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.

In summary, 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.

## 3.7 Parameters and arguments

Some of the 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 a definition for a 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(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to programmer-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`.

## 3.8 Variables and parameters are local

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

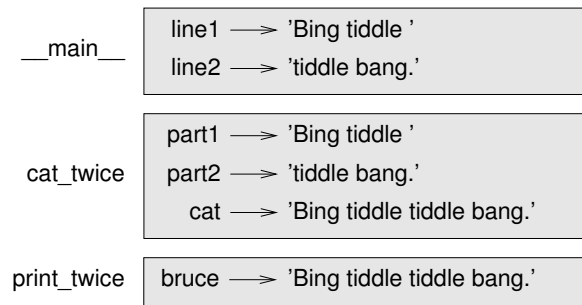


Figure 3.1: Stack diagram.

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside `print_twice`, there is no such thing as `bruce`.

### 3.9 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1.

The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

### 3.10 Fruitful functions and void functions

Some of the functions we have used, such as the math functions, return 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)
2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

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

This script computes the square root of 5, but since it doesn't store 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 assign the result to a variable, you get a special value called `None`.

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

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

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

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

### 3.11 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 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.

## 3.12 Debugging

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

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

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

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

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

## 3.13 Glossary

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

**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.

**body:** The sequence of statements inside a function definition.

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

**function call:** A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

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

**local variable:** A variable defined inside a function. A local variable can only be used inside its function.

**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.

**fruitful function:** A function that returns a value.

**void function:** A function that always returns None.

**None:** A special value returned by void functions.

**module:** A file that contains a collection of related functions and other definitions.

**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 values defined in a module.

**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.

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

**flow of execution:** The order statements run in.

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

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

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

## 3.14 Exercises

**Exercise 3.1.** Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
>>> right_justify('monty')  
monty
```

*Hint:* Use string concatenation and repetition. Also, Python provides a built-in function called `len` that returns the length of a string, so the value of `len('monty')` is 5.

**Exercise 3.2.** A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

```
def do_twice(f):
    f()
    f()
```

Here's an example that uses `do_twice` to call a function named `print_spam` twice.

```
def print_spam():
    print('spam')
```

```
do_twice(print_spam)
```

1. Type this example into a script and test it.
2. Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `print_twice` from earlier in this chapter to your script.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing 'spam' as an argument.
5. Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Solution: [http://thinkpython2.com/code/do\\_four.py](http://thinkpython2.com/code/do_four.py).

**Exercise 3.3.** Note: This exercise should be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```
+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

```
print('+', '-')
```

By default, `print` advances to the next line, but you can override that behavior and put a space at the end, like this:

```
print('+', end=' ')
print('-')
```

The output of these statements is '+ -' on the same line. The output from the next print statement would begin on the next line.

2. Write a function that draws a similar grid with four rows and four columns.

Solution: <http://thinkpython2.com/code/grid.py>. Credit: This exercise is based on an exercise in Oualline, Practical C Programming, Third Edition, O'Reilly Media, 1997.



## Chapter 5

# Conditionals and recursion

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: floor division and modulus.

### 5.1 Floor division and modulus

The **floor division** operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, rounding down:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

An alternative is to use the **modulus operator**, `%`, which divides two numbers and returns the remainder.

```
>>> remainder = minutes % 60
>>> remainder
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

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

If you are using Python 2, division works differently. The division operator, `/`, performs floor division if both operands are integers, and floating-point division if either operand is a float.

## 5.2 Boolean expressions

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

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

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

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

The `==` operator is one of the **relational 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
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. 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 relational operator. There is no such thing as `=<` or `=>`.

## 5.3 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 or both* 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`:

```
>>> 42 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 (unless you know what you are doing).

## 5.4 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 `if` is called the **condition**. If it is true, the indented statement runs. If not, nothing happens.

`if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**.

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

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

## 5.5 Alternative execution

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

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

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**, because they are branches in the flow of execution.

## 5.6 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 run. 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':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

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 runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

## 5.7 Nested conditionals

One conditional can also be nested within another. We could have written the example in the previous section 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. 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 runs only if we make it past both conditionals, so we can get the same effect with the `and` operator:

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

For this kind of condition, Python provides a more concise option:

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

## 5.8 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

If  $n$  is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs  $n$  and then calls a function named `countdown`—itself—passing  $n-1$  as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of `countdown` begins with  $n=3$ , and since  $n$  is greater than 0, it outputs the value 3, and then calls itself..

The execution of `countdown` begins with  $n=2$ , and since  $n$  is greater than 0, it outputs the value 2, and then calls itself..

The execution of `countdown` begins with  $n=1$ , and since  $n$  is greater than 0, it outputs the value 1, and then calls itself..

The execution of `countdown` begins with  $n=0$ , and since  $n$  is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The countdown that got  $n=1$  returns.

The countdown that got  $n=2$  returns.

The countdown that got  $n=3$  returns.

And then you’re back in `__main__`. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A function that calls itself is **recursive**; the process of executing it is called **recursion**.

As another example, we can write a function that prints a string  $n$  times.

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

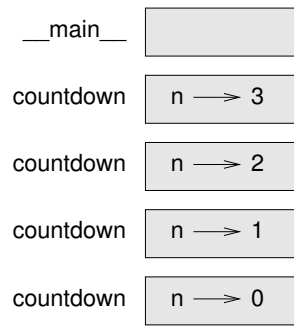


Figure 5.1: Stack diagram.

If  $n \leq 0$  the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display `s`  $n - 1$  additional times. So the number of lines of output is  $1 + (n - 1)$ , which adds up to  $n$ .

For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

## 5.9 Stack diagrams for recursive functions

In Section 3.9, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a frame to contain the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure 5.1 shows a stack diagram for `countdown` called with  $n = 3$ .

As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where  $n=0$ , is called the **base case**. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `print_n` called with `s = 'Hello'` and  $n=2$ . Then write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function `n` times.

## 5.10 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
.
.
.
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 recurse frames on the stack!

If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

## 5.11 Keyboard input

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that stops the program 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. In Python 2, the same function is called `raw_input`.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. `input` can take a prompt as an argument:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

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`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

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.
```

## 5.12 Debugging

When a syntax or runtime error occurs, the error message 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.

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

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

When you run this program, you get an exception:

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

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, which uses floor division instead of floating-point division.

You should take the time to read error messages carefully, but don't assume that everything they say is correct.



## 5.13 Glossary

**floor division:** An operator, denoted `//`, that divides two numbers and rounds down (toward negative infinity) to an integer.

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

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

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

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

**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 runs.

**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.

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

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

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

**return statement:** A statement that causes a function to end immediately and return to the caller.

**recursion:** The process of calling the function that is currently executing.

**base case:** A conditional branch in a recursive function that does not make a recursive call.

**infinite recursion:** A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

## 5.14 Exercises

**Exercise 5.1.** *The `time` module provides a function, also named `time`, that returns the current Greenwich Mean Time in “the epoch”, which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

*Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.*

**Exercise 5.2.** *Fermat’s Last Theorem says that there are no positive integers  $a$ ,  $b$ , and  $c$  such that*

$$a^n + b^n = c^n$$

*for any values of  $n$  greater than 2.*

1. Write a function named `check_fermat` that takes four parameters— $a$ ,  $b$ ,  $c$  and  $n$ —and checks to see if Fermat’s theorem holds. If  $n$  is greater than 2 and

$$a^n + b^n = c^n$$

*the program should print, “Holy smokes, Fermat was wrong!” Otherwise the program should print, “No, that doesn’t work.”*

2. Write a function that prompts the user to input values for  $a$ ,  $b$ ,  $c$  and  $n$ , converts them to integers, and uses `check_fermat` to check whether they violate Fermat’s theorem.

**Exercise 5.3.** *If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:*

*If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a “degenerate” triangle.)*

1. Write a function named `is_triangle` that takes three integers as arguments, and that prints either “Yes” or “No”, depending on whether you can or cannot form a triangle from sticks with the given lengths.
2. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses `is_triangle` to check whether sticks with the given lengths can form a triangle.

**Exercise 5.4.** *What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.*

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. What would happen if you called this function like this: `recurse(-1, 0)`?
2. Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).

The following exercises use the `turtle` module, described in Chapter 4:

**Exercise 5.5.** *Read the following function and see if you can figure out what it does (see the examples in Chapter 4). Then run it and see if you got it right.*

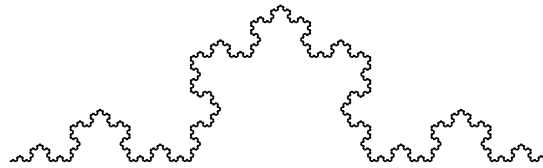


Figure 5.2: A Koch curve.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

**Exercise 5.6.** *The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length  $x$ , all you have to do is*

1. Draw a Koch curve with length  $x/3$ .
2. Turn left 60 degrees.
3. Draw a Koch curve with length  $x/3$ .
4. Turn right 120 degrees.
5. Draw a Koch curve with length  $x/3$ .
6. Turn left 60 degrees.
7. Draw a Koch curve with length  $x/3$ .

*The exception is if  $x$  is less than 3: in that case, you can just draw a straight line with length  $x$ .*

1. Write a function called `koch` that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.
2. Write a function called `snowflake` that draws three Koch curves to make the outline of a snowflake.

*Solution:* <http://thinkpython2.com/code/koch.py>.

3. The Koch curve can be generalized in several ways. See [http://en.wikipedia.org/wiki/Koch\\_snowflake](http://en.wikipedia.org/wiki/Koch_snowflake) for examples and implement your favorite.



## Chapter 6

# Fruitful functions

Many of the Python functions we have used, such as the math functions, produce return values. But the functions we've written are all void: they have an effect, like printing a value or moving a turtle, but they don't have a return value. In this chapter you will learn to write fruitful functions.

### 6.1 Return values

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

The functions we have written so far are void. Speaking casually, they have no return value; more precisely, their return value is `None`.

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This statement means: "Return immediately from this function and use the following expression as a return value." The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
    return math.pi * radius**2
```

On the other hand, **temporary variables** like `a` can make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these return statements are in an alternative conditional, only one runs.

As soon as a return statement runs, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if  $x$  happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0.

```
>>> print(absolute_value(0))
None
```

By the way, Python provides a built-in function called `abs` that computes absolute values.

As an exercise, write a compare function that takes two values,  $x$  and  $y$ , and returns 1 if  $x > y$ , 0 if  $x == y$ , and -1 if  $x < y$ .

## 6.2 Incremental development

As you write larger functions, you might find yourself spending more time debugging.

To deal with increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a floating-point value.

Immediately you can write an outline of the function:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
>>> distance(1, 2, 4, 6)
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 triangle. When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences  $x_2 - x_1$  and  $y_2 - y_1$ . The next version stores those values in temporary variables and prints them.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

If the function is working, it should display `dx is 3` and `dy is 4`. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use `math.sqrt` to compute and return the result:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the return statement.

The final version of the function doesn't display anything when it runs; it only returns a value. The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

As an exercise, use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments. Record each stage of the development process as you go.

### 6.3 Composition

As you should expect by now, you can call one function from within another. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

```
result = area(radius)
```

Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):  
    radius = distance(xc, yc, xp, yp)  
    result = area(radius)  
    return result
```

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):  
    return area(distance(xc, yc, xp, yp))
```

### 6.4 Boolean functions

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):  
    if x % y == 0:  
        return True  
    else:  
        return False
```



It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.

Here is an example:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

The result of the `==` operator is a boolean, so we can write the function more concisely by returning it directly:

```
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

But the extra comparison is unnecessary.

As an exercise, write a function `is_between(x, y, z)` that returns `True` if  $x \leq y \leq z$  or `False` otherwise.

## 6.5 More recursion

We have only covered a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

**vorpal:** An adjective used to describe something that is vorpal.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol `!`, you might get something like this:

$$0! = 1$$
$$n! = n(n - 1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other value,  $n$ , is  $n$  multiplied by the factorial of  $n - 1$ .

So  $3!$  is 3 times  $2!$ , which is 2 times  $1!$ , which is 1 times  $0!$ . Putting it all together,  $3!$  equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of  $n - 1$  and then multiply it by  $n$ :

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

The flow of execution for this program is similar to the flow of `countdown` in Section 5.8. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 2 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 1 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...

Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.

The return value, 1, is multiplied by  $n$ , which is 1, and the result is returned.

The return value, 1, is multiplied by  $n$ , which is 2, and the result is returned.

The return value (2) is multiplied by  $n$ , which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Figure 6.1 shows what the stack diagram looks like for this sequence of function calls.

The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not run.

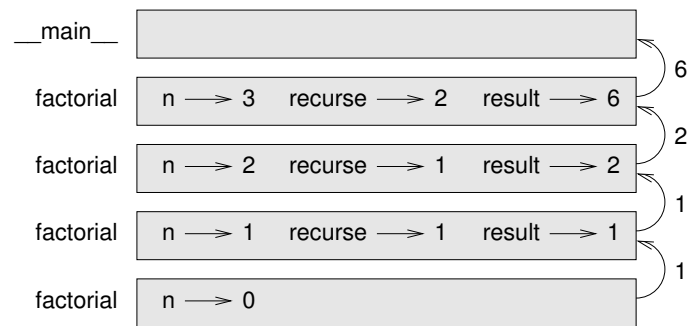


Figure 6.1: Stack diagram.

## 6.6 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the “leap of faith”. When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, “Assuming that I can find the factorial of  $n - 1$ , can I compute the factorial of  $n$ ?” It is clear that you can, by multiplying by  $n$ .

Of course, it’s a bit strange to assume that the function works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

## 6.7 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition (see [http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)):

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

Translated into Python, it looks like this:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of  $n$ , your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

## 6.8 Checking types

What happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case—when  $n == 0$ . But if  $n$  is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of  $n$  is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The first base case handles nonintegers; the second handles negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

If we get past both checks, we know that  $n$  is a non-negative integer, so we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

In Section 11.4 we will see a more flexible alternative to printing an error message: raising an exception.

## 6.9 Debugging

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting; a precondition is violated.
- There is something wrong with the function; a postcondition is violated.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

If the parameters look good, add a `print` statement before each `return` statement and display the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in Section 6.2).

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

Adding `print` statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with `print` statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(4)` :

```

        factorial 4
    factorial 3
    factorial 2
    factorial 1
    factorial 0
    returning 1
    returning 1
    returning 2
    returning 6
    returning 24

```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

## 6.10 Glossary

**temporary variable:** A variable used to store an intermediate value in a complex calculation.

**dead code:** Part of a program that can never run, often because it appears after a return statement.

**incremental development:** A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

**scaffolding:** Code that is used during program development but is not part of the final version.

**guardian:** A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

## 6.11 Exercises

**Exercise 6.1.** Draw a stack diagram for the following program. What does the program print?

```

def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

```

```
x = 1
y = x + 1
print(c(x, y+3, x+y))
```

**Exercise 6.2.** The Ackermann function,  $A(m, n)$ , is defined:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

See [http://en.wikipedia.org/wiki/Ackermann\\_function](http://en.wikipedia.org/wiki/Ackermann_function). Write a function named `ack` that evaluates the Ackermann function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of `m` and `n`? Solution: <http://thinkpython2.com/code/ackermann.py>.

**Exercise 6.3.** A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

We’ll see how they work in Chapter 8.

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `''` and contains no letters?
2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

Solution: [http://thinkpython2.com/code/palindrome\\_soln.py](http://thinkpython2.com/code/palindrome_soln.py).

**Exercise 6.4.** A number,  $a$ , is a power of  $b$  if it is divisible by  $b$  and  $a/b$  is a power of  $b$ . Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`. Note: you will have to think about the base case.

**Exercise 6.5.** The greatest common divisor (GCD) of  $a$  and  $b$  is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is based on the observation that if  $r$  is the remainder when  $a$  is divided by  $b$ , then  $\text{gcd}(a, b) = \text{gcd}(b, r)$ . As a base case, we can use  $\text{gcd}(a, 0) = a$ .

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor.

Credit: This exercise is based on an example from Abelson and Sussman’s *Structure and Interpretation of Computer Programs*.

