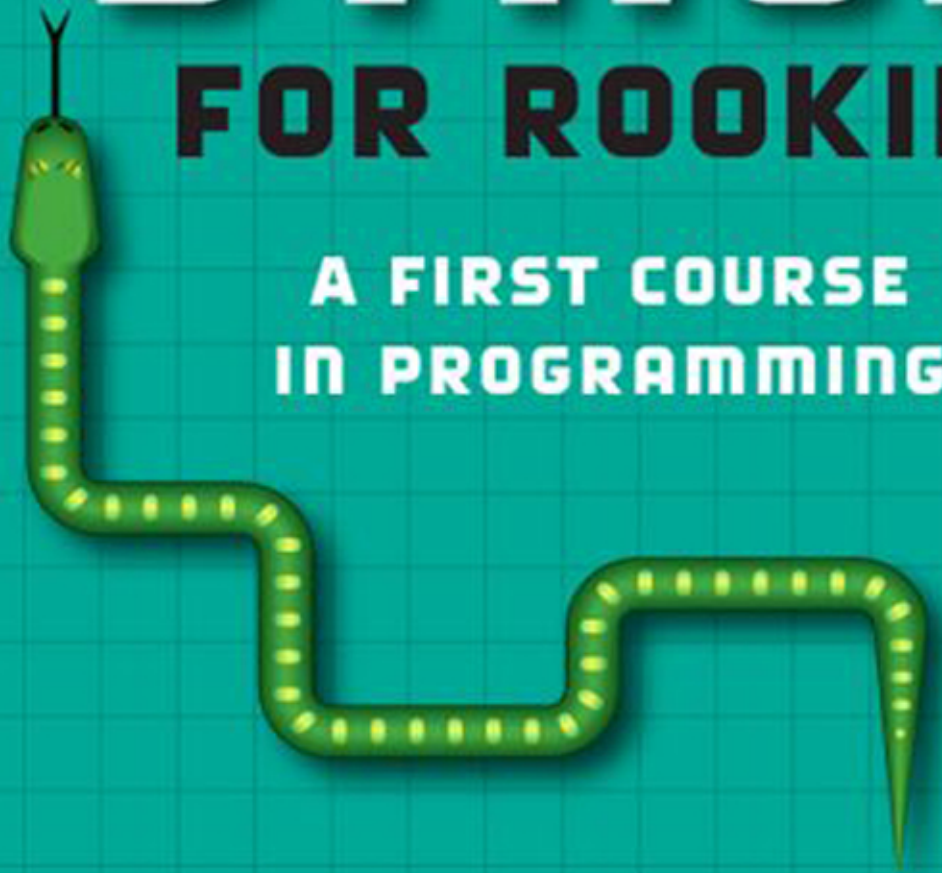
SARAH MOUNT
JAMES SHUTTLEWORTH
RUSSEL WINDER

PYTHON FOR ROOKIES

A FIRST COURSE IN PROGRAMMING



Visit the companion website at www.cengage.co.uk/python

Python for Rookies
A First Course in Programming
Sarah Mount, James Shuttleworth and Russel Winder

Publishing Director
John Yates

Publisher
Gaynor Redvers-Mutton

Editorial Assistant
Matthew Lane

Content Project Editor
Leonora Dawson-Bowling

Manufacturing Manager
Helen Mason

Marketing Manager
Jason Bennett

Typesetter
Russel Winder

Production Controller
Maeve Healy

Cover Design
Nick Welsh

Printer
C & C Offset Printing Co Ltd, China

For more information, contact Thomson Learning, High Holborn House; 50–51 Bedford Row, London WC1R 4LR or visit us on the World Wide Web at: http://www.thomsonlearning.co.uk

ISBN 978-1-84480-701-7

British Library Cataloguing-in-Publication Data A catalogue record for this book is available from the British Library.

1

Getting Started

Learning Outcomes

At the end of this chapter you will be able to:

- Use Python interactively.
- Evaluate arithmetic expressions using Python.
- Draw some simple pictures using the Python Turtle module.
- Create files containing Python programs and execute them.
- Import modules to be able to make use of facilities from the standard library.

These days, wherever electricity is available, there are computers. Although the term *computer* generally conjures up the idea of a desktop workstation or laptop, not all computers are that obvious. Almost all car engines now are controlled by one or more computers. Many household washing machines are controlled by a computer. From workstations and laptops, the Web, the Internet, televisions, set-top boxes, games consoles, cameras, washing machines, car engines, all the way down to smartcards, computers have become an integral and ubiquitous part of modern day living, at least in the first and second worlds.

Computers are not however complete systems in themselves: a computer by itself cannot actually do anything, it is just a lump of hardware that has potential. For a computer to be useful, it has to have programs to tell it what to do. The utility and usefulness of a computer depends entirely on the programs used: a computer is the hardware platform on which software directs the behavior of the complete system.

1.1 A Bit of Background

In the early days of computers (1940s–1950s) there were very few of them, they were used for academic, government or military purposes only and were programmed in *machine code* (the binary instruction language of a computer) or, if you were lucky, in a symbolic representation of machine code usually termed *assembly language*. Programming was hard and very few people did it.

As computers moved into the commercial world and the academics did research on how to program them (1950s–1970s), programming languages such as Lisp, Fortran, Cobol, Algol-60, etc., were created and used. The idea that people wrote programs and needed to understand them easily had taken hold. Time moved on,

more research was done, and we got programming languages like Pascal, Ada, C, C++. Inventing new programming languages became something of an industry. Of the many hundreds of programming languages invented only some became popular and used. For example, TCL, Perl, Objective Caml, Prolog, Haskell, Java, Ruby and, of course, Python have gained widespread adoption. There are many similarities and differences between all these programming languages and the histories of them and the inter-relationships between them are very interesting. For the moment, we only want to look at two particular issues (we will address many of the other issues as we go through the book).

1. When we write programs, we use a textual representation. Computers cannot understand the programs we write directly, they have to be translated (the official term is *compiled*) into a form the computer can work with. Programming languages like C, C++, etc. require programs to be compiled to the machine code of the computer on which the program is to be executed. Programming languages like Java, and, most importantly for this book, Python, use a *virtual machine*. A virtual machine is an idealized computer for executing programs. The *statements* of programs written in virtual-machine-based programming language are compiled into the machine language of the virtual machine, and there is then a mechanism to execute these virtual machine instructions on the real computer when the program is executed.

   You may well be asking yourself: "Why bother with virtual machines when we have real machines?" The answer has to do with being able to execute the same program on many different computers. With a language like C++, a program has to be compiled for each architecture and operating system. With languages such as Java and Python, a program compiled on one machine can be executed on another machine without recompilation. This is not a big issue for individuals working with a single machine, but it is a huge issue for organizations which have many computers, all needing to use the same programs – using virtual-machine-based languages means they do not have to ensure that all the computers are the same in order to be able to execute the program – the virtual machine mechanism does that for them.

2. When does compilation happen from the programmer's perspective? Programs in languages like C, C++, and Java must be compiled before they can be executed: an explicitly two stage process, compile then execute. Programs in languages like Ruby and Python can be executed directly without any prior explicit compilation step, as translation of the program occurs as needed behind the scenes.

Executing is what a program does when you double-click on its icon using a graphical user interface, or when you type a command in a command terminal – we will say more about what 'execute' means in Section 1.2, (page 3).

An *operating system* is the software that makes all the hardware work. Examples of operating systems are: Linux, Mac OS X, Solaris, and Windows.

So what does this mean for us now? 'Compile as needed' virtual-machine-based languages like Python involve significantly less hassle when creating and executing programs. This means Python programs are easier to work with than C++ or Java programs, particularly when first learning programming. Python is an (almost) ideal language for learning to program. Moreover, knowing how to program in Python is a great springboard for learning other programming languages like Ruby, Java and C++, so learning Python first really is a win–win situation.

A Note on Terminology

Programmers often talk about the *source code*, or sometimes just *code*, of a program. The terms source code and code really just mean all the statements that comprise the program in the language that the programmers are using.

1.2 Executing Programs

In computing, 'executing' is used in the sense of 'perform or carry out': executing a program means getting the computer to do what the program tells it to.

It may not be immediately obvious, but to be useful a program must, as a result of executing, do something that affects the world. It doesn't have to affect it greatly, it just has to affect it in some way. If it does not then the program serves no purpose.

Perhaps the simplest program in Python is:

```
print "Hello."
```

This has the effect, when executed, of writing something to the screen of the computer. It has affected the world in that it has presented something for the user to read. Not totally earth shattering but, after all, this is only our first program.

So what has this program shown us? Well, primarily that there are ways of outputting information for the user to read and there are ways of storing data. The print statement is the basic output action in Python and the sequence of characters enclosed in double quotes (i.e. "Hello.") is called a *string literal* or more usually just a *string*. A *literal* is a value represented directly in the programming language – the value to be used is literally what we typed in.

A competitor for the simplest Python program is:

```
print 1
```

which, when executed, outputs 1. In this program we have the integer literal 1 representing the integer value one. Comparing these two programs we see that the print statement is actually being quite clever: it works out for itself how to deal with the literal in order to print sensible things to the output.

The question on your lips is, of course: "How is the program executed?" The most immediate way of executing Python programs is to use the Python system interactively. This means starting the Python system in interactive mode and then typing the statements of your program at the Python prompt. How you do this depends on which operating system you are using.

- On Linux, Mac OS X, and Unix-like systems you type the command *python* at a command prompt and this starts the Python system in interactive mode. Of course, this assumes that Python is installed, which it is by default on almost all distributions of these operating systems. If, however, the command doesn't work on your computer, you will need to download and install Python. This is probably best achieved using the package manager for your operating system – for example: synaptic, aptitude or apt-get on Ubuntu; aptitude or apt-get on Debian; yum on Fedora or Red Hat; pkg-get or pkgadd on Solaris.

- On Windows systems, there are two possibilities:

  1. Use Cygwin, which includes Python. This works very much like the Linux and Mac OS X versions of Python – mostly because it is the same as those versions!
  2. Use the Python for Windows system. Installing this (from http://www.python.org) adds a menu entry in the start menu that has two alternate ways of running Python in interactive mode:

(a) A command prompt window with Python running. This behaves identically to the Cygwin, Linux, Mac OS X, or Unix versions of Python.
(b) An IDLE (**I**ntegrated **D**eve**L**opment **E**nvironment) window – see Figure 1.1.

IDLE is available on Linux, Mac OS X, Solaris and other Unix-like systems as well as Windows systems: just type *idle* at a command prompt – assuming IDLE is installed of course!

As the name **I**ntegrated **D**eve**L**opment **E**nvironment implies, IDLE is more than just a Python system running in interactive mode, it is a complete environment for developing Python programs.

We are going to avoid completely any suggestion that one of the two interactive systems is better than the other. It is really down to personal taste; some people prefer one, some the other. The important thing is to use whichever you prefer and find easier to create programs with.

For the following example, we have chosen to use Python in interactive mode running on an Ubuntu system:

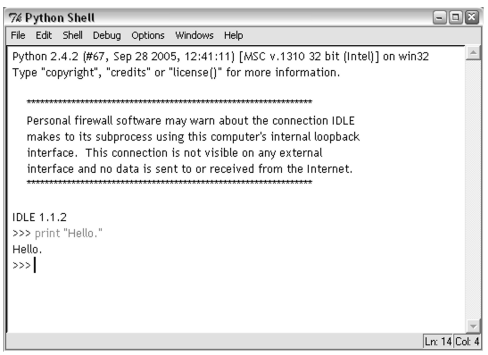
```
|> python
Python 2.5.1c1 (release25-maint, Apr 12 2007, 21:00:25)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello."
Hello.
>>>
```

The >>> is the interactive Python prompt. We typed the statement, pressed the return key and the statement was executed immediately. Note that the double quotes didn't get output. The double quotes in the program tell the Python system what the string is, they are not part of the string.

Things should look (more or less) the same whichever of the interactive versions of Python you try and certainly programs will behave the same on any Python system. This is a crucial point: Python looks and behaves the same on all computers. This sameness on all computers is one of the many benefits of using a system based on a virtual machine. Programmers can create programs in the full knowledge that the virtual machine takes care of all the differences between the actual computers being used. So we have portability of programs between different computers, exactly what we, as programmers, want.

Figure 1.1

The IDLE main window on Windows.



WORA Concept

Sun Microsystems use the (trademarked) phrase 'write once, run anywhere' about Java. The concept behind the phrase is as applicable to Python as it is to Java, perhaps more so.

1.3 And If Something Goes Wrong?

The two programs seen so far, being relatively simple (!), have no errors in them. Being error-free is an unusual situation for a program, but more on this later. What happens if there had been an error? The Python system would have told us so. For example, if we misspell **print** as **prin** then:

```
>>> prin 1
  File "<stdin>", line 1
    prin 1
         ^
SyntaxError: invalid syntax
>>>
```

The questions on the tip of your tongue are:

"What is *syntax*?"

and

"Why is the error at the 1 and not with the prin?"

Syntax is the way in which a statement in a language (any language, not just programming languages) is constructed.

Natural languages – languages with which humans communicate, for example English, Mandarin, Cantonese, Spanish – have syntactic rules and most speakers obey the rules. This is generally so that listeners, who assume the rules are being obeyed, can easily follow what is being said. Humans are, though, extremely good at dealing with syntactic errors in the use of language: humans use context, implicit knowledge, world knowledge and intelligence to construct a meaningful statement even in the face quite dramatic errors of syntax. It take concentration and effort, but it is done and often the meaning inferred (or guessed) is the meaning intended! Sometimes the wrong meaning is inferred and this can be the basis of much humor or the start of real, all to often violent, conflict. More usually though, visitors to a community, using a language they are not used to, can make truly awful errors in the use of the local language and yet still be able to communicate. People are good at language.

Programming languages are much, much simpler than natural languages and yet programming language compilers are usually extreme in their complete inability to deal with even the slightest divergence from the expected and allowed sequence of symbols. This is a very important point to bear in mind when programing in order to stay peaceful and sane. You have to remember that compilers are literal and, well, fundamentally stupid.

A nice quote, usually attributed to Richard Bornat, is: *Computers aren't stupid like an ant is stupid, computers are stupid like a brick is stupid.*

So, back to the program error. Why are we getting a syntax error in the statement:

```
prin 1
```

Humans who know a bit about Python will immediately say "Well **print** is spelt wrongly." However, the compiler sees the word prin but does not understand that it is a misspelling of **print**, so does not understand the word as **print** but as something else. It is only after the space when it sees the 1 that it realizes that what it is being asked to compile does not correspond with the allowed syntax of a Python statement. So it is only at the point of reading the 1 that the compiler appreciates that there is a syntax error. Compilers cannot perform any form of correction, so you get an error message and must correct the program and execute it again. This can be frustrating, but remember compilers are literal, they do not think like humans do. Actually they don't think at all!

There are two morals to this:

1. Whenever you get a syntax error, always look before as well as at the point marked to discover and understand what is wrong – it will eventually become second nature. As you get used to decoding error messages do remember that although the initial cause of the error may actually be your fault, the compiler is being very unhelpful by not deducing more about what you meant and giving sensible feedback.

2. Be patient in the face of error messages. At the risk of appearing a bit repetitive: remember, compilers are very literal and you have to do their thinking for them. Compilers are not like people when it comes to dealing with language syntax. More's the pity.

Memory

Memory in a computer is where everything gets stored during execution. It is usually measured in bytes. In the mid-1970s computers were lucky to have 16 KB of memory (16,000 bytes). In mid-2007 computers normally got delivered with at least 1 GB of memory (1,000,000,000 bytes).

1.4 Python can Calculate

Many people think of computers as just very complicated calculators, and in some sense this is true. Certainly computers know how to do their sums. For example:

Programming languages use * rather than × for multiplication as * is on the keyboard and × is not.

```
>>> print 4 + 9
13
>>> print 4 - 9
-5
>>> print 4 * 9
36
>>> print 4 / 9
0
>>>
```

Hmmm… that last one seems a bit wrong. The first three are as expected, but that last one… definitely not expected. Having said that, Python cannot have anything so basic wrong. Perhaps then the Python view of what the *expression* 4 / 9 means is different from what we initially assume it means? This is exactly right. When humans able to do sums see this expression, they know that the result is a fraction:

$$\frac{4}{9} = 0.44\dot{4}$$

where the dot over the last digit means that that digit recurs forever. So why does Python do something different?

For various historical reasons, computers do not treat numbers quite as humans and mathematics do. For most (but not all) computers, 4 is not the same as 4.0: 4 is an *integer*, whereas 4.0 is a *floating point* number – floating point numbers are numbers with both an integer part and a fractional part, and are usually called *floats* or sometimes *reals*. In mathematics, any integer number is a real number, so 4 and 4.0 are the same value. For computers, though, integers and floats are fundamentally different: integers and floats are represented differently in the hardware. So as an *expression* in Python, `4 / 9` says divide the integer `4` by the integer `9` and return an integer result. Since the actual result is a float with only a zero in front of the point, the integer result is `0`.

People sometimes use the term *decimal number* instead of floating point number, but this is not entirely useful in computing as decimal numbers can be either integer or floating point!

If instead of `4 / 9` we type `4.0 / 9.0` then we are asking Python to calculate the result of dividing the floating point number `4.0` by the floating point number `9.0`, giving a floating point result:

```
>>> print 4.0 / 9.0
0.444444444444
```

This is much more like it. Well, except that there is no indication that the fraction is a recurring one. This is a consequence of computers having finite amounts of memory. Since computer floating point numbers must be finite, infinite numbers cannot be represented in the computer. This means that computer floating point numbers are not exact but are just approximations to the actual values. For now we gloss over this, but we will have to return to the issue when we start doing some serious numerical computations.

Before moving on, we should try creating some more complex expressions to see whether anything else unexpected happens, or whether our understanding of evaluating arithmetic expressions from our mathematics education is reflected in Python. So:


```
>>> print 2 + 3 * 4
14
>>> print ( 2 + 3 ) * 4
20
>>> print 2 + 3 + 4
9
>>> print 2 + ( 3 + 4 )
9
```

This seems to indicate that the operations + and * behave as we expect them to: the *precedence* (aka priority) of the operations + and * are as we expect from mathematics. Also, parentheses, (and), seem to raise the precedence of the operation, again as we expect from mathematics.

The previous expressions used only integers: what about using floating point numbers?

```
>>> print 2.0 + 3.0 * 4.0
14.0
>>> print ( 2.0 + 3.0 ) * 4.0
20.0
>>> print 2.0 + 3.0 + 4.0
9.0
>>> print 2.0 + ( 3.0 + 4.0 )
9.0
```

BODMAS

BODMAS is an acronym used by many people to describe the precedence rule of arithmetic. It stands for:

Brackets
Orders
Division and ***M***ultiplication
Addition and ***S***ubtraction

Some people use the acronym *BIDMAS* – ***I***ndexes replacing ***O***rders.

The idea behind this acronym is to help you remember that when evaluating an expression, you first evaluate things in parentheses (brackets) then things raised to powers (exponentiation or orders) then multiplication and division in left-to-right order, and finally addition and subtraction in left to right order.

So the same holds true for floating point and integer numbers. We can do further experiments but if we do we will find all the results to be very much as expected. If you are not yet convinced yourself, try some really complicated expressions and see if you can get Python to do the wrong thing. We are confident it will be very hard!

It is not actually impossible to get incorrect answers but you have to work with very large numbers to get errors. Computer numbers are of finite size so there are numbers that cannot be stored.

Python can be very useful as a calculator, but there is clearly more to Python than that. Computers often, and workstations and laptops always, have screens for displaying things, so let's investigate drawing pictures on the screen.

1.5 Turtles can Draw

1.5.1 Libraries and Modules

One of the things that makes any programming language powerful is the amount of pre-written software that comes with it. Python has an extensive *library* of software that can be used by any programmer, at any time. This means that you don't have to write every program from scratch, you can make use of things in the library to do as much as possible. This is what makes Python so powerful.

One example is databases. The Python library comes with considerable support for creating and manipulating databases. This means that if we need to write a program to manage a database we don't have to write everything: we can make use of the library to do much of the hard work for us.

Another example is the support Python offers for user interface graphics which makes building user interfaces relatively straightforward. Actually, the real complexity of user interfaces is in the design and usability of the interface rather than in its construction. Python cannot help with those issues: they are human–computer interaction (HCI) issues.

In fact, the Python library has support for a huge range of software applications, helping you in almost any programming situation: communicating over the Internet, creating webpages, drawing pictures, in fact most things you can think of.

Rather than having the whole of the library available all the time, the library is structured as a set of *modules*. Each module is self-contained and so can be used independently of other modules. This means that we can tell the Python system to use just the parts needed for a particular program. The **import** statement is the way we tell the Python system that we need to make use of a module.

Logo, The Programing Language

In 1966 Seymour Papert and Wally Feurzeig designed *Logo*, a simple language for studying some ideas in artificial intelligence. Logo was also used for programming 'turtles'. The earliest turtles were radio-controlled robots which had a pen, a bell, touch sensors and could move forward, turn right, turn left, lower the pen, raise the pen, etc. Logo evolved into a language for drawing graphics on a computer screen, in particular for moving a turtle symbol simulating the behavior of the robot turtle. This turned out to be a fun language with which to write programs. Perhaps more importantly, it was very easy for beginners, including young children, to learn to program. This was shown experimentally first by Feurzig at Hanscom Field School, Lincoln, Massachusetts, USA, and Papert at Muzzy Junior High, Lexington, Massachusetts, USA and later by Papert at various other schools and institutions – they used Logo very successfully for teaching children about algorithms and programming.

When Guido van Rossum created Python, he included a module (the Turtle module) that realized in Python the core ideas of the Logo language. People find this Python module equally simple and fun to use, so we use it in various places in this book.

1.5.2 Using the Turtle Module

In this section, we use a module called Turtle to draw simple pictures by moving a virtual robot (aka *turtle*) around the screen. Having started the Python system, we have to issue the command **import** turtle to request the Python system to do whatever is necessary to import the Turtle module:

```
Python 2.5.1c1 (release25-maint, Apr 12 2007, 21:00:25)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import turtle
```

Now we can do something with the Turtle module. First let's try the standard demonstration to get the turtle to show us what it can do. We do this by typing turtle.demo ( ) after the import statement:

```
>>> import turtle
>>> turtle.demo ( )
```

Figure 1.2 shows the result of executing the statements. As is clear from the figure, the turtle can draw lines of different thickness, create shapes, fill those shapes and write text. What is not so obvious from the figure (given that it is monochrome) is

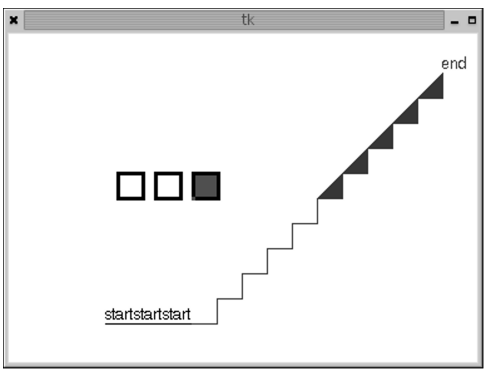


Figure 1.2

Result of running the demo function in the Turtle module. Some of this is colored red.

that various parts of the image are in different colors. If you run the demonstration yourself, you will see all the colors involved.

The statement we entered to run the turtle demo is clearly different from the **print** and **import** statements. Firstly, we used the name turtle.demo, i.e. we gave the name of the module followed by a period (aka full stop) then the name of the action. Secondly, we appended an open and close parenthesis, i.e. we typed turtle.demo ( ) rather than turtle.demo. turtle.demo is an example of a *function* and we append the parentheses to indicate that the function should be executed. If we had not put the parentheses, we get:

```
>>> turtle.demo
<function demo at 0xb7ac72cc>
>>>
```

which tells us that turtle.demo is a function that resides at some location in the memory of the computer.

"What is a function?" you are asking. A function is a way of collecting together a sequence of statements and giving it a name, so that the sequence of statements can be executed without having to type them all out. Indeed we don't even have to know what the sequence of statements is to be able to execute it, we just have to know what the name is – as with the turtle.demo function above.

Modules can have many functions: the Turtle module has many for drawing different shapes.

1.5.3 Lines and Directions

Let's now get the turtle to draw something for us. Obviously it is a good idea to clear the window first. We do this by calling the function turtle.reset, which clears the window as shown in Figure 1.3.

```
>>> import turtle
>>> turtle.demo ( )
>>> turtle.reset ( )
```

Now we have a clear screen, we can draw some shapes of our own. Let's start with something simple, a square:

```
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
```

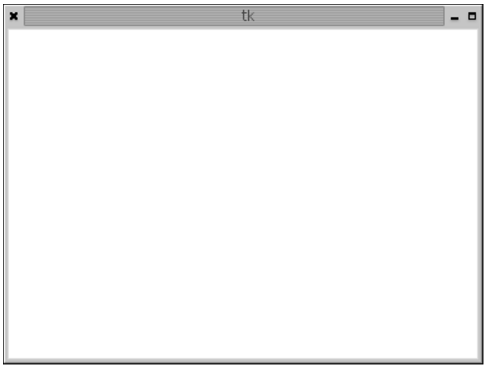


Figure 1.3

Result of calling turtle.reset.

```
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
```

The result is shown in Figure 1.4.

To draw the square, we have used two functions, turtle.forward and turtle.left, that have integers in the parentheses of the function call. The turtle.forward and turtle.left functions require *parameters* (the numbers in the parentheses). For turtle.forward, the parameter tells the system how many units to move by – 100 in all the examples above. The units are *pixels*. Computer displays (just like televisions) are screens that are composed of a two dimensional array of pixels. Typical sizes for computer displays are 800×600, 1024×768, 1280×1024, 1600×1200, 1920×1200. High-definition television is 1368×768. (Display sizes for LCD displays are relatively simple, but things get really rather complicated for television. See, for example, http://en.wikipedia.org/wiki/Display_resolution.) For turtle.left, the parameter tells the system by how much to turn the turtle left. By default this parameter is in degrees. So turtle.left (45) means turn anticlockwise by 45°.

'pixel' is an abbreviation of 'picture element'

Measuring Angles

Angles can be measured in *radians* but most people learn about degrees and not radians, so the Turtle module works with degrees by default. We can work with radians if we want, we just have to call turtle.radians.

As you might already have guessed, the Turtle module also provides functions for moving backwards (turtle.backward) and turning clockwise (turtle.right), along with many other drawing functions.

1.5.4 Some Other Functions in the Turtle Module

We claimed in the caption of Figure 1.2 that parts of that diagram were drawn in red even though it does not appear so in this monochrome book. Hopefully you have run the demo to prove to yourself that this claim is actually true! So the turtle can change the color of the pen used for writing. Color changes are achieved by calling the turtle.color function. The change only applies to drawing done after the

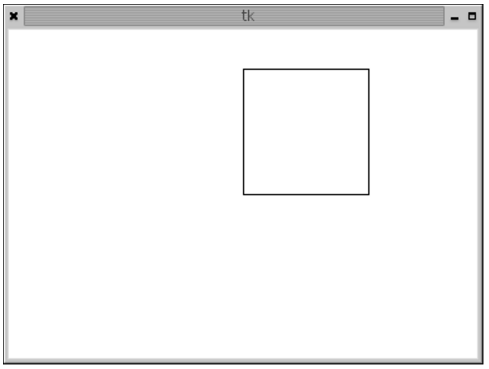


Figure 1.4

Drawing a square.

call to the turtle.color function, so we need to draw something new to see that the change of color has happened:

```
>>> turtle.reset ( )
>>> turtle.color ( "Red" )
>>> turtle.forward ( 10 )
```

The result is shown (in monochrome) in Figure 1.5.

The turtle can also raise and lower the pen just as if we were raising and lowering a real pen on a piece of paper – this harks back to the real physical turtle controlled by Logo programs. Unsurprisingly, these functions are called turtle.up and turtle.down. Moving with a raised pen leaves no trace while, as we have already seen, moving with a lowered pen draws a line (lowered is the default position). Clearly, we can use turtle.up and turtle.down to draw a dashed line:

```
>>> turtle.up ( )
>>> turtle.forward ( 10 )
>>> turtle.down ( )
>>> turtle.forward ( 10 )
>>> turtle.up ( )
>>> turtle.forward ( 10 )
>>> turtle.down ( )
>>> turtle.forward ( 10 )
```

Figure 1.6 shows the result of executing this program.

Figure 1.5

Drawing in a new color. As the book is monochrome the line looks grey, but if you try this for yourself, you will see it is red.

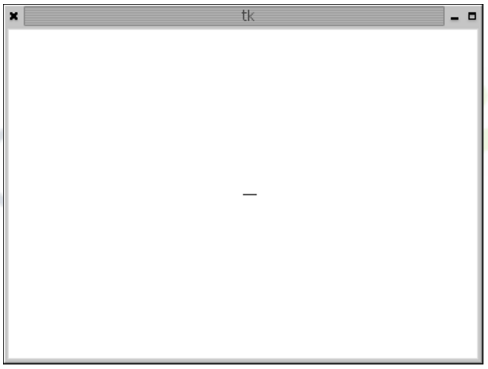
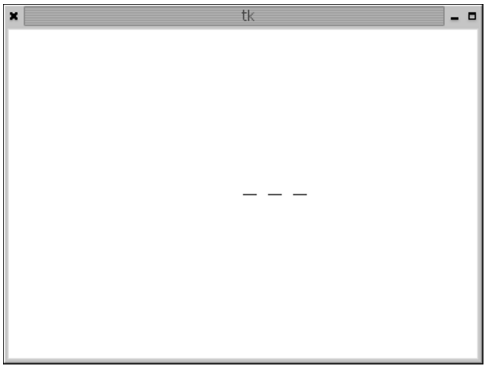


Figure 1.6

Drawing a dashed line.



The Turtle module has the turtle.circle function which, unsurprisingly, draws circles. So, for example, typing:

```
>>> turtle.reset ( )
>>> turtle.circle ( 50 )
```

gives the result shown in Figure 1.7. The parameter to the turtle.circle function specifies the radius (in pixels) of the circle to be drawn. The turtle starts drawing in the direction it is facing and draws a circle of the specified radius ending in the position and direction in which it started.

The turtle.circle function can draw circular arcs as well as complete circles: we can give calls of turtle.circle a second parameter that specifies how much of a circle to draw. By typing:

```
>>> turtle.reset ( )
>>> turtle.circle ( 50 , 180 )
```

we get only a half a circle, as shown in Figure 1.8. When we call the turtle.circle function with only one parameter, the Python system assumes we are telling it to draw all 360° of the circle. Calling the function with a second parameter overrides this assumption and draws only part of the circle; in this example half a circle (180°). Circles are always drawn by traveling anti-clockwise, i.e. the center of the circle or circular arc being drawn is always to the turtle's left of its current position.

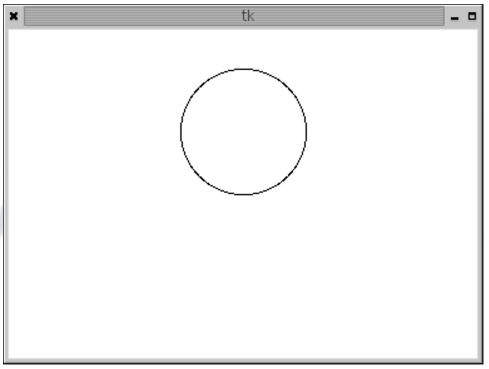


Figure 1.7

Drawing a circle.

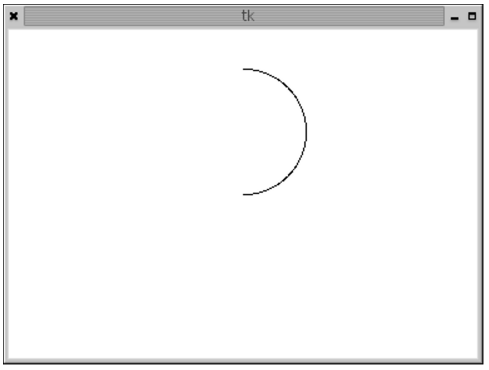


Figure 1.8

Drawing half a circle.

Help

If at any time during an interactive session you want to find out what functions are available in a module, you can enter help mode by executing the help function:

```
>>> help ( )

Welcome to Python 2.5! This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics". Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

At this prompt we can type the name of a module, for example 'turtle'. The output is quite lengthy so we will not reproduce it here. Give it a try and you will see information about the module, including what functions are available. Quite a lot of the information presented may only make sense after you have a little more experience of Python, but there is also information there that we want, such as the list of functions we can use.

The best way of finding out what each of the functions can do and what can be achieved by combining them is to experiment – although we probably ought to use the term 'play' here rather than 'experiment', since 'play' implies far more strongly that programming the turtle can be fun!

1.6 Algorithms

An *algorithm* is a sequence of instructions that can be used to carry out a task. You have already come across algorithms in your daily life: for example, if you ask someone how to get from your home to the nearest shop selling console games, the answer will be an algorithm, perhaps something such as 'go down the road, turn left at the traffic lights, take the third right and the shop is on your left'. It is important that you follow the instructions in the correct order. If you try to take the third right before you turn left at the traffic lights, you may well get very lost indeed!

Algorithm, The Word

Algorithms are named after the ninth century Persian mathematician Muhammad bin Musa al-Khwarizmi. Originally, *algorisms* were rules for performing arithmetic using Arabic numerals. By the eighteenth century, 'algorism' had become 'algorithm' and had come to mean a sequence of commands to carry out a specific task.

Augusta Ada King née Byron, Countess of Lovelace
The First Computer Programmer

The first algorithms written for a computer were Ada Byron's programs for Charles Babbage's Analytical Engine, written in 1842. Ada Byron was therefore, arguably, the first computer programmer. The language Ada was named after her as recognition of her contribution to computing.

A computer program is an algorithm expressed in a programming language so that it may be executed by a computer. As with algorithms in any context, it is important that the statements of the program are executed is the correct order. For example, if we try to call turtle.reset before we tell Python to import the Turtle module, we get this error:

```
Python 2.5.1c1 (release25-maint, Apr 12 2007, 21:00:25)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> turtle.reset ( )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'turtle' is not defined
>>>
```

The Python system is telling us that it doesn't have anything called turtle to refer to, so it can't run the turtle.reset function for us.

More often, though, getting an algorithm wrong does not result in an error message, just unexpected results. Let's take the program for drawing a square of side 100 pixels:

```
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
```


and rearrange it slightly so that it is wrong (we swap the order of the last two statements):

```
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 100 )
>>> turtle.forward ( 100 )
>>> turtle.left ( 90 )
```

Figure 1.9 shows the result. Definitely not a square, but there is no error message, as there is no error as far as the Python system is concerned. The program we gave the Python system is a perfectly reasonable program and it executed it as requested. As the intention was to draw a square, the program clearly has one or more errors, but not ones the computer can detect. There is nothing wrong with the syntax of the program, it is either that the algorithm is wrong or that the correct algorithm has not been correctly implemented, so the result is not what was intended. In this case, our instructions to Python, which were intended to mean

'draw a square', resulted in an unexpected result because the code was wrong – we had the right algorithm, but wrote the statements in the wrong order. Such an error is called (obviously) an *algorithmic error* or (less obviously) a *semantic error*.

In the case of algorithms such as that for drawing a square, it is usually fairly obvious where errors are if they occur. It is then easy to correct the program. However, for anything but the smallest and simplest of algorithms this is not generally the case. Semantic errors can be hard to find and sometimes even harder to correct. It is extremely important, therefore, to plan your programs carefully and make sure you know what they *ought* to do before you write them.

Always understand what the program you are writing is intended to do *before* you write it.

Later in the book (Chapter 9) we will introduce a way of programming, Test-Driven Development, that really helps avoid semantic errors. However, we need a little more programming and Python technology before we can introduce that. So for the moment we just have to be very careful.

The terms bug *and* debug

Grace Murray Hopper was an early pioneer of computing. She invented the compiler in 1952 and later the programming language COBOL. She used to tell a story about a technician who fixed a 'bug' in the Harvard Mark II computer by pulling an insect out of the contacts of one of its relays. Since then, the terms *bug* meaning a mistake in a computer program and *debug* meaning to fix a bug have become widely used. For many years the logbook associated with the incident and the bug in question (a moth) were displayed at the Naval Surface Warfare Center (NSWC).

However, this may not be the origin of the term. It is reported that the term bug was used in Thomas Edison's time (late nineteenth century) to mean a fault in electrical apparatus. It is even reputed to have been a term used during the development of wireless telegraphy earlier in the nineteenth century!

1.7 Designing an Algorithm: Square Spirals

Figure 1.10 shows a square spiral. To show an example of an algorithm (and of developing a Python program), we shall set ourselves the task of creating a Python program to draw such a square spiral using the Turtle module.

The first thing we have to do is fully understand the result we want to achieve. This might seem obvious, but it's amazing how many software projects fail simply because the developers writing the software never really fully understood what their client wanted.

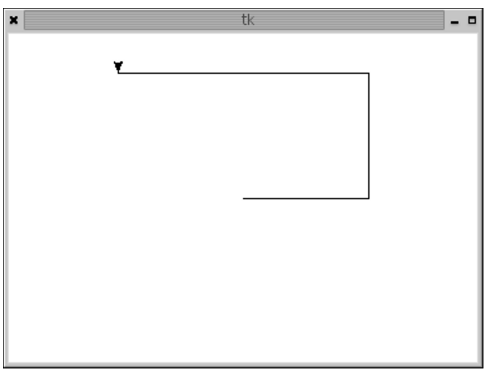


Figure 1.9

Not a square but it was supposed to be!

Looking at Figure 1.10, we see that a square spiral is made up of straight lines joined together at right angles, almost a square but not quite. Looking at the center of the example and tracing it out, we see that all the joins of this example are left-hand turns. So, to draw this figure, our program is going to be made up of calls to the `turtle.forward` and `turtle.left` functions. As all the joins are right-angles, the turtle will always need to turn left by 90° at the end of each straight line.

Have we mentioned what a good idea it is to understand what the program you are writing is intended to do *before* you write it?

So what distinguishes a square spiral from a square? Tracing out the square spiral from the center, we see that the straight lines get longer. In this example, we draw two lines of the same length, then increase the length of the next pair of lines, and so on. If we increase the line length too much, the shape will spiral quickly. If we increase the side length too little then we will spiral out slowly. Of course if we do not increase the length at all then there is no spiral, we end up drawing a square over and over again!

So, having thought about how to create the shape we want, we can think about creating a program. In this case thing are fairly simple: we can trace out the example and it tells us the algorithm and hence the code to write. Actually, this is often the easiest way of writing programs like this: draw the shape on a piece of paper and it tells you what the algorithm is and hence what code is needed to achieve that shape.

This:

is three sides of a square, with each side the same length. If we trace round this shape (starting from the lower left-hand corner) we see that, because the third side is the same length as the other two, the next side to be drawn will meet the first one and form a square. We don't want that, so the *third* side needs to be longer than the first two, to make a spiral. So, our square spiral algorithm will be something like this:

Move forward by some amount
Turn left 90°
Move forward by some amount
Turn left 90°
Move forward by a bit more than last time
Turn left 90°
Move forward by new amount
Turn left 90°
Move forward by a bit more than last time
Turn left 90°
Move forward by new amount
...

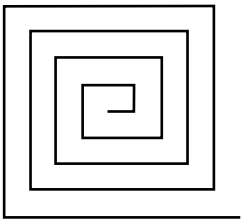


Figure 1.10

A square spiral.

This way of expressing algorithm – using 'clipped' natural language – is called *pseudocode*. Pseudocode is very much program oriented but it is not any particular programming language, nor does it have any rules. The idea is that we express the algorithm in a programming language independent way but using language that is easily translatable into any programming language.

To write our Python program we simply need to add Python-specific detail to the pseudocode expression of the algorithm. Let's say that we will start by drawing a line of length 10 and that we will increase the line length by 10 when the length to move forward by is increased. We are now in a position to write a sequence of Python statements that implements this algorithm:

```
>>> import turtle
>>> turtle.forward ( 10 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 10 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 20 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 20 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 30 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 30 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 40 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 40 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 50 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 50 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 60 )
>>> turtle.left ( 90 )
>>> turtle.forward ( 60 )
>>> turtle.left ( 90 )
```

which results in the drawing shown in Figure 1.11.

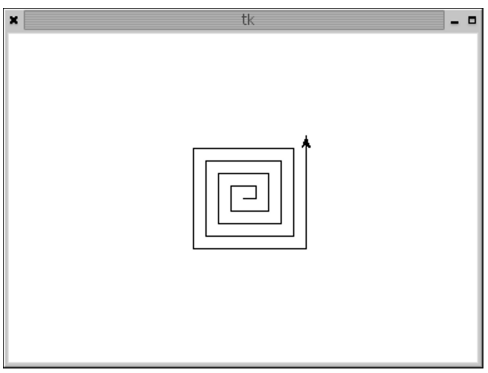


Figure 1.11

A square spiral drawn with the turtle.

1.8 Storing and Executing Programs

You are probably thinking: "But if I want to draw the shape again, I have to type it all in again. How tedious." or even "If I want to compare two slightly different sequences, I have to type almost the same thing more than once. Irritating." So, whilst typing programs in line by line and having each statement executed as soon as we press the return key is all very instant and can be incredibly useful, it is absolutely clear that we need to be able to prepare our programs before executing them. Obviously, we need to store programs in some way independent of the Python system that we can then execute as Python programs. In fact, this is the usual way of programming: use a text editor to create a file and then get the Python system to execute that file. So for example we can use a text editor (we use XEmacs but any text editor will do) to put the statement:

Always use a text editor for editing program code. Word processors such as OpenOffice.org and Word are not appropriate because they are intended for creating documents not programs.

```
print "Hello."
```

into a file, we called our file printHello.py, and then we can then issue the command *python printHello.py* to execute the program:

```
|> python printHello.py
Hello.
|>
```

Issuing the Python command without a file name puts the system into interactive mode while giving it a file name causes the system to open the file and execute the statements in the file. Many people call this *script mode* to distinguish it from interactive mode.

The Python system is fundamentally the same whichever mode we use it in. In particular, any program executed in either way will behave the same and all errors are the same and reported in the same way. For example, the faulty Python program:

```
prin 1
```

when executed, results in:

```
|> python helloError.py
  File "helloError.py", line 1
    prin 1
         ^
SyntaxError: invalid syntax
```

which is exactly (well almost exactly) the same message as we got in interactive mode – the difference is that the name of the file is given instead of '<stdin>'. As far as the Python system is concerned all input and output happens through files. The file could be a real file on disk or it could be a 'special' file which is actually connected to a human input device, for example the keyboard. The file associated with the keyboard is called 'stdin', hence the error message from earlier said the input was from '<stdin>' – the '<' and '>' are there to show it is a special file, and not a file on disk. The Python compilation and execution system does not distinguish the two cases, a file is a file as far as it is concerned. So using the Python system in either script mode or interactive mode really is the same.

Chapter Summary

In this chapter we have found that:

- Python programs are sequences of statements.
- Python handles arithmetic and arithmetic expressions very much as expected, except for the issue of integer and floating point numbers.
- Python has a large library, structured as a set of modules.
- Python modules contain functions which we can call to do various things.
- The Python system has an interactive mode and a script mode.
- Python programs are usually constructed using a text editor as files on disk, and then executed in script mode.

Self-Review Questions

Self-review 1.1 Why is the following the case?

```
>>> print 4 + 9
13
>>>
```

Self-review 1.2 Why is the following the case – rather than the result being 13?

```
>>> print "4 + 9"
4 + 9
>>>
```

Self-review 1.3 Why does the expression 2 + 3 * 4 result in the value 14 and not the value 24?

Self-review 1.4 What is a module and why do we have them?

Self-review 1.5 What is a library and why do we have them?

Self-review 1.6 What does the statement `turtle.demo ( )` mean?

Self-review 1.7 What is an algorithm?

Self-review 1.8 Why are algorithms important in programming?

Self-review 1.9 What shape does the following program draw?

```
turtle.forward ( 100 )
turtle.right ( 90 )
turtle.forward ( 50 )
turtle.right ( 90 )
turtle.forward ( 100 )
turtle.right ( 90 )
turtle.forward ( 50 )
```

Self-review 1.10 The Turtle module provides a function to draw circles. If this function was not available, what algorithm would you use to draw a circle?

Self-review 1.11 What does the function turtle.goto do?

Hint: Use the Python help system to help you!

Self-review 1.12 What is stdin?

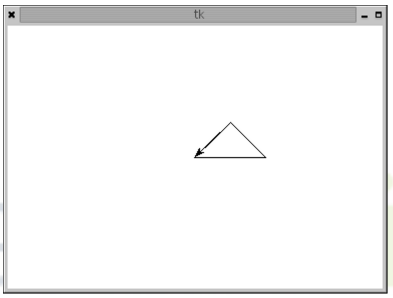
Self-review 1.13 What is 'script mode'?

Programming Exercises

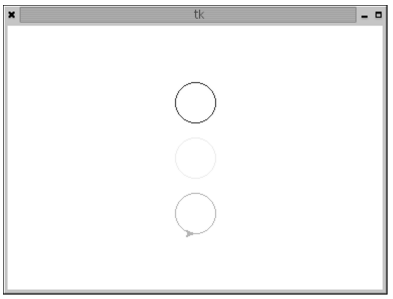
Exercise 1.1 Estimate the size of your computer screen by importing the Turtle module and causing the turtle to move 1 pixel (though you may want to try 10 pixels!).

Exercise 1.2 Experiment with the 'square spiral' program, varying the lengths of various lines. Which values make spirals and which do not? Which values make the nicest-looking spiral?

Exercise 1.3 Write a program to draw a right-angled triangle looking something like:



Exercise 1.4 Write a program to draw a red circle, then a yellow circle underneath it and a green circle underneath that. The result should look something like:

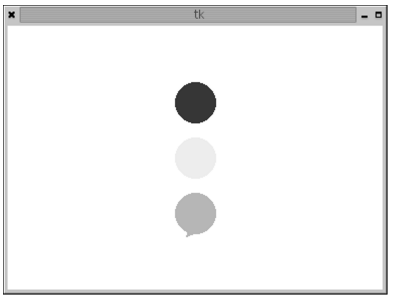


It is true that in this monochrome rendition, the colors just appears in various shades of grey, but the colors will work on your computer – unless you have a monochrome screen!

Hint: We used the function turtle.goto as well as the functions turtle.up, turtle.down, turtle.color and turtle.circle in our solution.

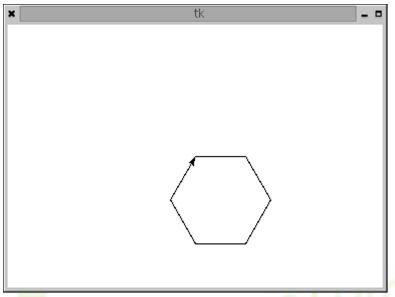
Exercise 1.5 Amend your program from the previous question so that the circles are filled and hence the result looks something like:

These circles are definitely filled with the right color when we run the program on our computers even though they just look greyish in this book.



Hint: We used the function turtle.fill in our solution.

Exercise 1.6 Write a program to draw a regular hexagon. The result should look something like:

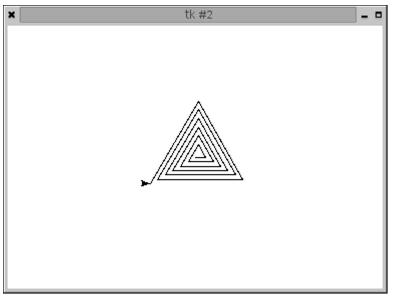


Hint: Regular hexagons have six sides, of equal length, which meet at an internal angle of 120°.

Exercise 1.7 Write a program to draw the first letter of your name.

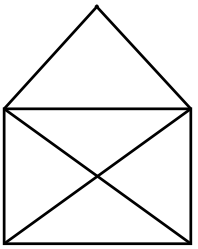
Challenges

Challenge 1.1 Write a program to create a triangular spiral, as in:



Hint: This is easiest if you work with equilateral triangles – it gets very complex otherwise.

Challenge 1.2 Write a program to draw this 'envelope' figure:



Hint: You may want to start by doing some trigonometry to sort out all the angles and lengths. Pythagoras' Theorem will almost certainly come in useful: Pythagoras' Theorem states that for a right-angled triangle:

$$hypotenuse = \sqrt{x^2 + y^2}$$

where hypotenuse *is the longest side of the triangle and* x *and* y *are the lengths of the other two sides.*

Hint: To find the square root of a number in Python, you need to import the module **math** *and call the function* **math.sqrt***: to find the square root of 2, you import the* **math** *module, then write* **math.sqrt(2)***.*

Challenge 1.3 Draw the following figure: