Introduction to Programming In Python

Brian S. Blais

Acknowledgments

I want to thank the Bryant College AI and Robotics class of Fall 2003 for having the patience with me teaching the course *without* this guide. Their good work has motivated me to write it. The Bryant University AI and Robotics class of Fall 2004 allowed me to clean up many of the exercises, and make the text more clear. The AI and Robotics class of 2011 motivated me to do many improvements to this document.

Preface: Why Python?

There are several reasons for me choosing Python as the language/environment for introducing programming, some of them practical, some pedagogical, and some personal.

- Python is free, as in "free oil changes". I believe this is an important consideration for teaching any language. If it is accessible, then students are more likely to use it beyond the particular class that is being taught.
- Python is free, as in "free speech". This allows me to distribute freely anything I develop with Python, and any changes to Python that I make for my own purposes.
- Python is available both for Windows, Linux, and the Mac. Since I work in both a Mac and a Linux environment, but am surrounded by students who only know MS Windows, this is a personal reason for me using it.
- Python is simple. It's syntax is extremely clean. There are no variable declarations or pointers. There are no segmentation faults or other memory errors.
- Python is interpreted. This makes it easy to develop programs, and more importantly to *debug* programs.
- Python is flexible. You can do dynamic heterogeneous arrays, structures, lists, objects and classes with a simple syntax. You can do both procedural and object-oriented programming (or a mixture) if you want.

This guide grew out of my experience teaching an Artificial Intelligence and Robotics course at Bryant College. The course includes students from many majors, not many of which are computer oriented. As such, Python was a choice (over the traditional Lisp for AI) that would have a chance to be used by students in other contexts outside of the class.

As far as other possible languages for the course, C or Java are obvious choices, especially given how much support material there is out there (I wouldn't have had to write this guide at all), but I believe that certain language structures, like variable declarations and pointers (in C), get in the way of the concepts I want to teach. Visual Basic is a possibility, but I don't know that language as well as others, I don't believe it exists currently for Linux, and I don't believe it has the simplicity and power of Python. Lisp is the most common choice for an AI class, but as noted above the course I teach is not a traditional Computer Science AI course. Pascal is not portable enough, and Prolog is just too weird. :)

Since running my classes with Python for many years now, I feel I have accomplished things in this language that would be very difficult, or impossible, to do with a different language. Such things include using Python to program Lego Mindstorms robots, writing numerical simulators with near-mathematical notation, and flexibly dealing with many different types of data. I have yet to have a project where Python wasn't the easiest language to develop that project.

Contents

Chapter 1

Introduction

This guide is written to provide an introduction to programming, for those who may have no programming experience. It uses a language/environment called Python (available for free online at `www.python.org`), although for Windows I prefer the Enthought (`www.enthought.com`) version. At the time of writing this guide, the current version of Python is 2.7 (with the Enthought version). The guide includes example code and exercises for the reader (with solutions for the instructor). Programming is a skill which can only be learned by doing, so the examples covered later in the guide depend on code written as exercises earlier on. It is therefore important to do all of the exercises along the way.

1.1 What is Programming?

A *program* is a sequence of instructions telling the computer what to do to accomplish some task. Programming amounts to determining the proper sequence of instructions to accomplish the task, writing these instructions in some language (also called computer *code*), and testing the results of running the program to make sure that the program is in fact accomplishing the task correctly. When people write these programs for computers to read and execute, difficulties arise because computers read languages very differently than people read languages. There are two main qualities of computers which are important here:

1. Computers are *syntactically picky*.

   Syntax refers to the rules about about is allowed in a language. For example, the following is *not* syntactically correct English:

   Th.e qui'ck br.own f/ox ju[mps ov]er t=he 1azy blog.

   There are rules in English for the placement of punctuation marks which are seriously violated in this "sentence" (to call it a sentence is to infer that it is syntactically correct). Unless you are really observant, you may not have even noticed the difference between "lazy" and "1azy", the latter having the digit "1" (one) in front. As humans we can overlook these small syntactical problems and still infer the meaning of the sentence (or *most* of the meaning, if we refuse to assume we meant "dog" instead of "blog"). The computer cannot do this. Any small syntactical error will cause the program to fail.

2. Computers are *completely literal*.

   Computer languages have *unambiguous* rules: each statement has one, and only one, meaning. One example of this we will encounter later is in translating an English sentence like "x is less than y and z" into code. What we *mean* is "x is less than y and x is also less than z". Depending on how we translate the code, the computer could interpret the sentence as "(x is less than y) and z" which, in more proper English, would be "z is a true statement and, in addition, x is less than y". Quite a different meaning than what we intended!

Solving the syntax problems are the easiest, because the computer will generally tell you where they are (even if the error messages it gives are a bit obtuse). Solving the problems in *meaning* or what I refer to as the *logic* of the program, is much harder. It is very common for a programmer to stare at the computer swearing that it is not doing "what I told it to do", when in fact the computer does *exactly* what you tell it to do. It may not be doing what you *meant* it to do. Tracking down these problems, a process known as *debugging*, takes time and practice.

Learning to program is like learning to play a musical instrument. You can read all you want, you can watch others do it, but until you program the computer yourself you will never really learn how to do it.

1.2 Installation

Download and Install

The easiest way to download python, for use in scientific areas, is to use the Enthought Edition. The link is `http://www.enthought.com/products/edudownload.php`. You then select your operating system, and click download to download the Academic Version. If you are asked in the installation process whether you have the commercial or academic version, choose academic (unless you want to pay some money!).

Extra Steps for Windows

The result of your installation, in Windows, will be two icons on your desktop: one called Pylab and the other called Mayavi. We won't use Mayavi, so you can delete it from your desktop. To get Python ready to be used for this guide, do the following:

1. Right-click on the *Pylab* link, and choose *Copy*
2. Right-click on the *Desktop* and choose *Paste*
3. Rename the *Copy of Pylab* to *Python*
4. Right-click on this new *Python* shortcut, and look at the *Target:* field, which should look like:
5. Delete the `-pylab` part of the *Target:* field, so it looks like:
6. Click *Ok*

This the *Python* shortcut you will use for the bulk of this guide.

Now do the following:

1. Open some folder, any folder
2. Go to the *Tools* menu, and select *Folder Options*
3. Scroll down to the checkbox called *Hide Extensions for Files with Known FIle Types*
4. Make sure that the box is *unchecked*
5. Click *Ok*

This is an annoying default setting in Windows which hides useful information.

You will need a text editor, better than Notepad, so do the following to install *Notepad++*:

1. Go to `http://notepad-plus-plus.org/download`
2. Select *Download the current release*
3. Click on the option for the *Installer* (at current writing it looks like *Notepad++ v5.8.6 Installer*)
4. Install the Editor

Once the editor is installed, there are a couple of settings that need to be adjusted.

1. Open *Notepad++*
2. Under the *Settings* menu then *Preferences*, choose *Language Menu/Tab Settings*
3. Make sure that the *Tab Size* is set to *4* and that the *Replace by Space* box is *checked*

Extra Steps for the Mac

You will need a text editor, better than the default, so do the following to install *TextWrangler*:

1. Go to `http://www.barebones.com/products/textwrangler/download.html`
2. Double-click on the `TextWrangler_3.1.dmg` to get the disk image
3. Double-click on the *TextWrangler* disk, and drag the TextWrangler icon to your *Applications* folder

Once the editor is installed, there are a couple of settings that need to be adjusted.

1. Open *TextWrangler*
2. Under the *TextWrangler* menu then *Preferences* (Command-,) make sure that the checkbox *Auto Expand Tabs* is *checked*

1.3 What is Python?

Python is, as stated in the official introduction,

> ... an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

When you run python you will will be greeted with the *command window* which will allow you to enter commands on the *command line*. All actions in Python are typed on the command line.

```
Enthought Python Distribution -- http://www.enthought.com

Python 2.6.6 |EPD 6.3-2 (32-bit)| (r266:84292, Sep 23 2010, 11:52:53)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

Everything in this tutorial which is written in `this font` is the text displayed in the command window or in a Python program. When I give commands to type on the command line, like the command for help, I will display it like this

```
In [3]: help()
```

Notice that the only part that you would type is the `help()` part, not the `In [3]:` part which is just the prompt given by Python to say that it is awaiting a command.

At minimum, the Python interpreter is a fancy calculator. For example,

```
In [4]:365*454
Out[4]:165710

In [5]:43.0/324.0
Out[5]:0.13271604938271606

In [6]:2**10
Out[6]:1024

In [7]:import math  # import the math functions

In [8]:math.sin(5)
Out[8]:-0.95892427466313845

In [9]:math.cos(3.14159)
Out[9]:-0.99999999999647926

In [10]:2+2+2+2+2
Out[10]:10

In [11]:(3+2)*(6+5)
Out[11]:55
```

All of the arithmetic operators (+, -, *, /, **) are supported. To use the standard math functions, like the trigonometric functions (sin, cos, tan), one needs to import the math module, with

```
In [7]:import math
```

After that, one can use them, preceding them with `math.`, like `math.sin`, `math.cos`, etc...

1.4 Your First Program: Hello World

It is programming tradition to have your first program, in whatever language, simply print out a message saying "Hello, World!". Although it is perhaps the most basic program to write, in order to get it to run you will already have to be able to do several steps:

1. Edit the program
2. Save it in the proper place
3. Run the program

I will now outline these steps. The standard programming set up is an editor and a command line, as shown in Figure 1.1.

Edit the program

When we write code, we use a text editor to type in the commands and then we save the file with a `.py` extension. In fact, I always start a new program by saving a *blank file* with the proper name, in this case `hello.py`. This way the editor knows I am working with a python file and colors my text nicely. Open up the editor, save as `hello.py` in your work folder, type the lines below, and then make sure to save it again.

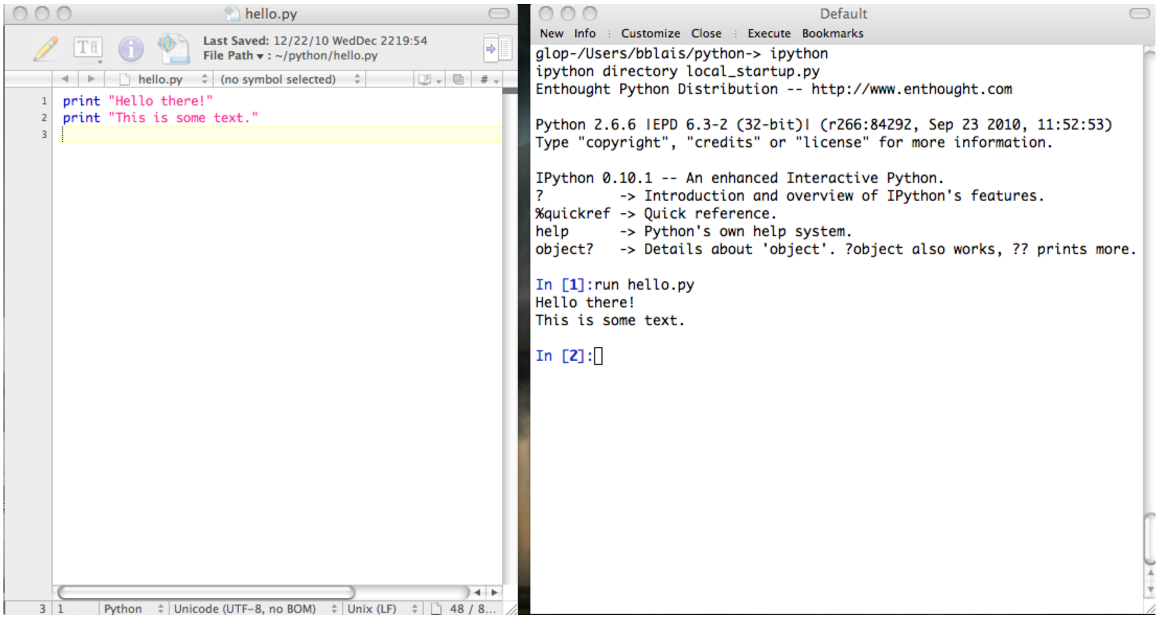


Figure 1.1: Standard programming set up. On the left is the editor, editing a program, and where the program is saved to disk. On the right is the command line where the program is actually run, from the last saved version.

```
print "Hello, World!"
```

Save it in the proper place

Save the file in a directory for all of your Python work.

Run the program

Now that our program has been saved, all we need to do is run it. This is done on the command line like:

```
In [2]:run hello.py
Hello, World!
```

You've just written your first Python program! When executing programs, like `hello.py`, Python executes each line in the program in order.

Error!

You may find the following error happens:

```
In [1]:run hello.py
ERROR: File `hello.py` not found.
```

There are two main reasons for this error.

1. Typo. Perhaps you meant `hello2.py` or something else? Check that you spelled the filename right!
2. Wrong folder. This is most common, at the beginning, when students are new to python. It happens when python is working in a folder, and you've saved the file to a *different* folder. Check this with the `pwd` command:

```
In [2]:pwd
Out[2]: '/Users/bblais'
```

Ah ha! I've saved the file in `/Users/bblais/python`, but python is working in `/Users/bblais` so it can't see it! Double-check where you saved the file.

A Slightly More Complex Program

A slightly more complex "hello world" program (saved as `hello2.py`) is the following:

```
print "Hello, World!"
print "The result of 2+2 is",2+2
```

The output is

```
In [1]:run hello2.py
Hello, World!
The result of 2+2 is 4
```

One More Example Program

```
import random

print "Hello, World!"
a=random.randint(1,10) # random number from 1-10
b=random.randint(1,10)  # random number from 1-10

c=a+b

print "The result of",a,"+",b,"=",c
```

The output is

```
In [3]:run hello3.py
Hello, World!
The result of 1 + 4 = 5
```

Try running it several times! What does it do?

> **Exercise 1.1**
>
> Type in each of the `hello` programs, and run them several times. Do they do the same thing each time?

Chapter 2

Program Structure - Part I

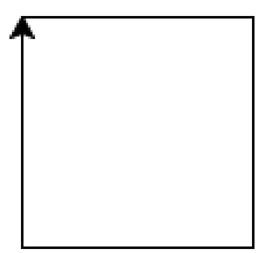
2.1 The Turtle

A very nice way to introduce program is using the `turtle` module. This module is a graphics module that lets you instruct a so-called "turtle", giving it instructions to go forward, turn right, go backward, etc... while using a pen to draw its path. It's easiest explained with an example.

Listing 2.1: "Drawing a Square"

```
# import the turtle functions
from turtle import *

forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
```

...which draws...



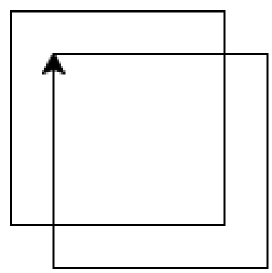
The directions tell the turtle to go forward for 100 pixels, turn right 90 degrees, go forward 100 pixels, etc... All the while the pen is down, so that the turtle draws. We can extend this example, by lifting the pen, moving over a little (without drawing), dropping the pen down, and drawing another square.

Listing 2.2: "Drawing a Two Squares"

```
# import the turtle functions
from turtle import *

forward(100)
right(90)
```

```
forward(100)
right(90)
forward(100)
right(90)
forward(100)

penup()
backward(20)
right(90)
forward(20)
pendown()

forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
```

...which draws...



Exercise 2.2

Make the turtle draw triangles, instead of squares, in the figure above.

Want to know more about the turtle?

The formal documentation is available at `http://docs.python.org/library/turtle.html`. Many of the functions are easily understood just from their names. Some of the more useful commands are:

- `forward(distance)` - Move the turtle forward by the specified `distance`, in the direction the turtle is headed.
- `backward(distance)` - Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
- `right(angle)` - Turn turtle right by angle units. (Units are by default degrees)
- `left(angle)` - Turn turtle left by angle units. (Units are by default degrees)
- `setheading(to_angle)` - Set the orientation of the turtle to `to_angle`.
- `goto(x, y)` - Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.
- `position()` - Return the current x,y coordinates of the turtle

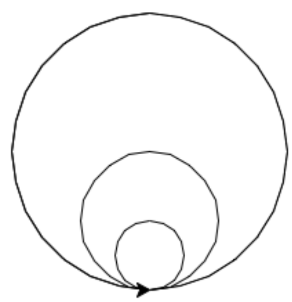
```
>>> turtle.pos()
(440.00,-0.00)
```

- `speed(0)` - Make the turtle draw very quickly
- `reset()` - Reset the screen and the turtle position. Used at the beginning of a script to make sure that running it again won't overlap the drawings.
- `circle(radius)` - Draw a circle of given `radius`, in a counter-clockwise direction from the current turtle heading. *The turtle is* **not** *the center of the circle.* To get a circle with the center at the turtle, you'll need to move the turtle over by radius units, draw the circle, and then move back.

Listing 2.3: "Drawing Circles"

```
from turtle import *
speed(0) # make the turtle fast

# draw a series of circles
circle(100)
circle(50)
circle(25)
```

...which draws...

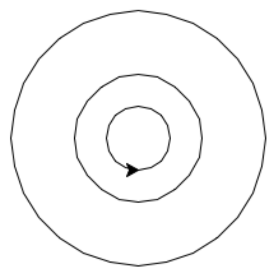


Listing 2.4: "Drawing Circles Centered at the Origin"

```
from turtle import *
reset()
speed(0) # make the turtle fast

# draw a series of circles centered at 0,0
penup()
right(90)
forward(100)
left(90)
pendown()

circle(100)

penup()
left(90)
forward(50)
right(90)
pendown()

```

```
circle(50)

penup()
left(90)
forward(25)
right(90)
pendown()

circle(25)
```

...which draws...



- `pencolor(color)` - Set the pen color
  - `pencolor(colorstring)` - Set pencolor to colorstring such as "red", "yellow", or "#33cc8c".
  - `pencolor(r, g, b)` - Set pencolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..1.

2.2 Variables

All data in a program is stored in *variables*, which are just memory blocks with names. The names can be any sequence of letters, underscores (_), or numbers as long as the name starts with a letter. `bob`, `frank4`, and `a_5_b` are all legitimate variable names. Usually you choose names that make sense for your particular application, like `mysum`, `total`, or `chicken3`. In Python, everything is *case-sensitive*. This means that `bob` is a *different variable* than `Bob`, `BoB`, or `boB` (which are all different from each other).

To assign a value (or values, as we shall see later) into a variable one uses the assignment operator, namely the "=" sign.

```
In [1]:a=5

In [2]:a*600
Out[2]:3000
```

In the last line I used `a` (which was defined to be `5` in the previous line) in an expression. Variables can be used just as numbers in any expression in Python.

```
In [6]:a=30

In [7]:b=40

In [8]:import math    # import the math functions

In [9]:c=math.sqrt(a**2+b**2)

In [10]:print c
50.0
```

where `math.sqrt` is a function that takes the square root of the numbers.

One can even use a variable in an expression which sets its own value.

```
In [11]:a=6

In [12]:b=7

In [13]:a=a+b

In [14]:print a
13
```

This demonstrates the meaning of the symbol `=` in Python. It does *not* mean the same thing that it means in standard algebra. It does not mean equivalence, it means *assign*. `a=5` means take the right side (`5`) and assign it to the variable on the left side (`a`). With `a=6` and `b=7`, then the statement `a=a+b` means take the right side (`a+b` which evaluates to 13) and assign it to the variable on the left side (`a`). Now `a` has the new value of 13.

Some of the uses of variables is for convenience, readability, consistency, and calculation. Let's consider our square program from Listing 2.1 on page 7. There are two obvious issues with it. One is that we've repeated ourselves several times. There is an easier way to make use of that, which we'll discuss later. The other issue is that the same number, the side of the square, is used four times. If we wanted to draw a square of a different size we'd have to change all four numbers. Because we have to make four changes, instead of one, it leads to more possible errors (i.e. typos). A variable can change that.

Listing 2.5: "Drawing a Square Using Variables"

```
# import the turtle functions
from turtle import *
reset()

size=100

# draw the square
forward(size)
right(90)
forward(size)
right(90)
forward(size)
right(90)
forward(size)
```

In the code, notice the use of *comments* following the `#` character. Python ignores anything following this characters, which allows us to document the code making it easier for others to read. Get in the habit of putting in comments now, because the biggest use of comments is to remind *yourself* about how a piece of code works, several weeks after it has been written.

2.3 User Input

Any good program asks for input from the user. There are two straightforward ways of doing this. The `input` function gets input from the keyboard, asked at the command line. This lets the user type in values, which can be different each time the program is run. There will be many cases where these values may be inappropriate, and possibly cause the program to crash. Good program writing will include taking care of these cases as well. The following example lets the user enter the size of a square to draw.

Listing 2.6: "Ask a User for the Size of the Square - Command line version"

```
# import the turtle functions
```

```
from turtle import *

reset()

# use a the command line to ask for the size
size=input('What size do you want the square?')

# draw the square
forward(size)
right(90)
forward(size)
right(90)
forward(size)
right(90)
forward(size)
```

When performing graphical tasks, one can use a graphical user interface (GUI) command, like in the following case.

Listing 2.7: “Ask a User for the Size of the Square - GUI version”

```
# import the turtle functions
from turtle import *
from tkSimpleDialog import askinteger

reset()

# use a pop-up window to ask for the size
size=askinteger('Square Size','What size do you want the square?')

# draw the square
forward(size)
right(90)
forward(size)
right(90)
forward(size)
right(90)
forward(size)
```

Exercise 2.3

Write a program to ask the user for the size, and the x and y coordinates of the center of a circle, and draw it.

The following example lets the computer determine a number you have chosen, given the answers to a couple of simple questions.

```
# simple guess the number game
#

import time   # the time module includes the sleep function

print 'Please think of a number between 10 and 99.'

sum_digits=input('What is the sum of the digits of your number? ')

```

```
diff_digits=input('What is the difference of the digits of your number? ')

print 'Let me think a moment...'

time.sleep(2); # delay for 2 seconds

# apply the equation to get the guess from the info
guess=10*(sum_digits+diff_digits)/2.0+(sum_digits-diff_digits)/2.0

print 'Your number was ',guess,', right?'
```

The following example computes the amount of interest earned after one year, given an initial principal and an annual interest rate.

```
# get the initial values
principal=input('What is the initial principal? ')
rate=input('What is the annual interest rate? ')

print 'The original principal is $',principal
print 'The interest rate is ',rate

interest=principal*rate # calculate the interest
print 'The total interest after 1 year is $',interest

principal=principal+interest
print 'The new principal after 1 year is $',principal
```

Note how the use of relevant variable names makes the code much easier to read.

2.4 Branching: If-statements

Programs execute one line at a time, in consecutive order. This sequential program flow can be modified using two types of programing structures: branches and loops. We consider branches in this section, and loops in the next section.

The form of a simple branch is the following:

```
if CONDITION:
    STATEMENTS
```

where I am using `ALL CAPITAL LETTERS` to denote something which stands for code that you would need to write, but is not code itself. The colon (:) is necessary at the end of the `if` line, or a syntax error will result. A branch, or if-statement, is interpreted in the following way: only execute the `STATEMENTS` if the `CONDITION` is true. For example, here is a program which swaps the two values of `x` and `y` only if `x` is originally larger than `y`. The result is that the final value of `x` will always be smaller, or equal to, the final value of `y`.

```
# An example which swaps two variable values,
# only if the first is larger

# get the initial values
x=input('What is the initial value of x? ')
y=input('What is the initial value of y? ')

# display the initial values
print 'x has the value ',x
```

```
print 'y has the value ',y
print '------------------------------'

if x>y:
    print '***Swapping***'
    x,y = y,x

# display the final values
print 'x has the value ',x
print 'y has the value ',y
```

Say we input the values of 5 for `x` and 10 for `y`.

```
What is the initial value of x? 5
What is the initial value of y? 10
x has the value  5
y has the value  10
------------------------------
x has the value  5
y has the value  10
```

As the program progresses, it gets to the line `if x>y:` which is calculated to be `if 5>10:` or `if False:` since `5` is not greater than `10`. Because the condition is false, the statements after the `if` which are indented by 4 spaces will not be executed, and the program continues with the next line of code after the indented lines. If, however, we input the values of 15 for `x` and 10 for `y`.

```
What is the initial value of x? 15
What is the initial value of y? 10
x has the value  15
y has the value  10
------------------------------
***Swapping***
x has the value  10
y has the value  15
```

The values get swapped!

Comment on Boolean variables

A boolean variable is one that has a value of true or false, instead of a number. In Python those values are `True` and `False` (note the capital first letter). Thus, each of the following if-statements will print a message to the screen

```
if True:
    print 'this line gets executed'

if 3>2:
    print 'this line gets executed'

printit= (3>2)  # set the variable printit to True

if printit:
    print 'this line gets executed'
```

whereas the following if-statements will print nothing

```
if False:
```

```
    print 'this line does not get executed'

if 50<10:
    print 'this line does not get executed'

printit= (34>57)  # set the variable printit to False

if printit:
    print 'this line does not get executed'
```

The `CONDITION` in an if-statement *must* reduce to a true or false value for it to have any meaning.

Boolean Operators

The following are the allowed boolean operators.

`==`	equal to
`>`	greater-than
`<`	less-than
`>=`	greater-than or equal to
`<=`	less-than or equal to
`not`	not (negation)
`!=`	not equal to
`and`	and
`or`	or

A **very** common mistake for beginning programmers is to use the assignment `=` instead of the equality `==` in a `CONDITION` of an if-statement. Whenever you are testing if two values are equal in if-statement (or a while-loop, which we discuss later), you **must** use `==`. For example, the following code fragment will print out a message

```
a=5
b=5
if a==b:
    print 'a is equal to b'
```

but the following code fragment will not

```
a=5
b=6 # <--- b is not equal to a
if a==b:
    print 'a is equal to b'
```

If one wanted to test to see if `x` is bigger than both `y` and `z`, one would write

```
if x>y and x<z:
    print x is bigger than both y and z
```

Translating back to English we have "`x` is greater than `y` and `x` is also greater than `z`". What would have happened if we had written instead, `x > y and z`? If you test it yourself you will find that it prints the message whenever `z` is not equal to zero, and `x` is greater than `y`! Why is that? As stated in the beginning, computers are very literal, and follow a strict syntax. Python is interpreting `x > y and z` as `(x>y) and (z)` where `z` is seen as a *boolean* (true/false) variable even though we didn't mean it to. In this sense, anything non-zero is true, so Python is interpreting `x > y and z` as true whenever `z` is not equal to zero, and `x` is greater than `y`.

Although it is not *always* necessary, it is a very good habit to put parentheses around any operation of two variables, like `(x>y)` or `( (x>y) and (x>z) )`. It may be a bit more typing, but it can save you hours in debugging logic that is hard to see otherwise.

if, elif, and else

The if-statement has a more general structure which is very useful. It looks like

```
if CONDITION1:
    STATEMENTS  # these statements run if CONDITION1 is true
elif CONDITION2:
    STATEMENTS  # these statements run if CONDITION1 is false, and
                #                         CONDITION2 is true
elif CONDITION3:
    STATEMENTS  # these statements run if CONDITION1 is false, and
                #                         CONDITION2 is false, and
                #                         CONDITION3 is true
else:
    STATEMENTS  % these statements run if all of the CONDITIONS are false
```

You can have as many or as few (even zero) `elif` clauses, and either include or not the final `else` clause. This structure lets you set up different actions for many different incoming possibilities. The colon (:) needs to be at the end of each line which starts a block of code, like `if`, `elif`, and `else`.

For example, the following program asks the user if she likes bananas, and responds differently given the user's response. Although we haven't used strings so far, the example is fairly self-explanatory.

```
#  raw_input, as opposed to input,  tells Python to input a string
# (of characters) rather than a number as a response

response=raw_input('Do you like Bananas? ');

if response=='yes':
    print 'I like Bananas too!'
elif response=='no':
    print 'I dislike Bananas too!'
else:
    print 'I did not understand what you wrote.'
```

Exercise 2.4

Make a turtle program to ask the user what shape to draw, and draw it. You should have at least 3 different shape choices.

2.5 Loops

A loop is used to repeat a set of statements many times, usually until some condition is met. One loop structure we will introduce now is the while-loop. It has the form

```
while CONDITION:
    STATEMENTS
```

This structure works like a repeating if-statement: if `CONDITION` is true, then the `STATEMENTS` are executed. In a while-loop, however, the program flow returns back to the `while (CONDITION)` line and the `CONDITION` is tested again. If it is still true, then the `STATEMENTS` will be executed *again*. This will repeat until such time as `CONDITION` is tested and comes up false. Then program flow jumps to the line following the entire `while` clause. The following is an example that prints out the numbers 1, 2, 3, 4 and 5.

```
x=1
while x<=5:
```

```
    print x
    x=x+1

print 'Done!'
```

Executed like:

```
>> run first5.py
1
2
3
4
5
Done!
```

The program flow in this example is as follows. The first line to be executed is `x=1` which sets the value of `x` to 1. Then the program tests to see if `x` is less than or equal to five, which is true, so the lines within the while-loop are executed. The first line displays `x`, which prints a "1" on the screen, and the next line adds one to `x`, yielding the answer 2, and assigns this new value to `x`. The program then jumps back to the `while` statement and tests `x` again to see if it is less than or equal to five, which is again true. Again, the value of `x` is printed to the screen, this time it is "2", and again `x` is incremented by 1, yielding 3. The `while` tests `x` again at 3, 4, and 5, passing each time and displaying the result. When `x` passes with a value of 5, the two lines are executed, displaying "5" and incrementing `x` to 6. The `while` tests `x` to see if it is less than or equal to 5, which is *false* now, and skips to the indented lines in the `while` block. The next line to be executed displays "Done!" and the program ends. Notice that the last value of `x` is 6, but the last value to be displayed is 5.

We can extend the `interest` program written in Section 2.2 to use a while-loop, and calculate the accumulated interest over the course of many years.

```
# get the initial values
principal=input('What is the initial principal? ')
rate=input('What is the annual interest rate? ')
number_of_years=input('How many years do you want to calculate interest? ')

print 'The original principal is $',principal
print 'The interest rate is ',rate

current_year=1
while current_year<=number_of_years:

    interest=principal*rate; # calculate the interest
    principal=principal+interest;

    print 'After year ',current_year,': '
    print '  The interest is $',interest
    print '  And the new principal is $',principal

    current_year=current_year+1;
```

One technique for using a while loop is to repeat a question if the user gave a bad answer. For example, the following code fragment keeps the user from entering a negative principal value, but allows the user to retype a valid answer.

```
principal=-1

while principal<0:
    principal=input('What is the initial principal? ')
```

```

    if principal<0:
        print 'The principal value cannot be negative.  Please reenter it.'
```

It is necessary for the initial value of **principal** to be less than zero, so that the statements within the while-loop will execute the first time. If we forgot the `principal=-1` line, we'd receive an error like:

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

If we had set `principal` to a positive value, then the while-loop would have tested *false*, and skipped all of the statements in the while-loop. The `principal=-1` line gets us into the while loop. After that, the user input will keep us there until the user enters a valid, positive (or zero) principal.

Exercise 2.5

Acey-Ducey is a game where you draw 2 cards, and then bet on whether the next card drawn will lie between the first two in rank. If the next card does lie between the first two in rank, then you win. Write a program called `acey_ducey` which let's you start with $100, draws two cards (numbers from 1-13), asks for a bet, draws a third card, and adjusts your money to reflect the win or loss. Continue until you lose all your money or enter a negative bet (to signal that you are done playing). Make sure you can't bet more than the amount of money you have!

The `for`-loop

Ninety percent of loops one writes, repeat a specified number of times, like the first example above, which repeats 5 times. Because of this, there is a more convenient form of a loop for this purpose, called a `for`-loop. The following two pieces of code do the same thing:

```
# while-loop
x=0
while x<10:
    print x
    x=x+1

print 'Done!'

# do the same thing with a for-loop

for x in range(10):
    print x

print 'Done!'
```

The `for`-loop moves through each value of `range(10)`, which goes from 0 to 9 (not 1 to 10), which repeats the statements in the `for`-loop 10 times. Both the setting of the initial value, and the incrementing, is done automatically.

The following example is the same as the Listing 2.1 on page 7, but uses a `for`-loop to reduce some of the redundancy.

Listing 2.8: "Draw a square using a for-loop"

```
# import the turtle functions
from turtle import *
reset()

size=100
```

```
for side in range(4):
    forward(size)
    right(90)
```

Another example with a square, with some extra variables. What do these variables do?

Listing 2.9: "Draw a square using a for-loop with some extras"

```
# import the turtle functions
from turtle import *
reset()

number_of_sides=4
angle=360/number_of_sides
size=100

for side in range(number_of_sides):
    forward(size)
    right(angle)
```

2.6 Functions

Functions are the basis of programming: all of the commands that you use are functions. So, what is a function?

Think of a function as a box, with information that you put into it, and information that it sends out, but you don't know how the insides of the box work. When you call the `math.cos` function, for example, you give it an angle, and it returns a number between -1 and 1. You don't know how Python actually implements `math.cos`.

When organizing code, you break the code up into three types of functions:

1. functions that *only* display information (and return nothing)
2. functions that *only* ask for user input (and return it)
3. functions that *only* calculate information, but display *nothing* (and return the result)

When I use the word "return" here, think again of the `cos` function. If I call:

```
>>> y=math.cos(0)
```

you will notice that nothing is displayed. The value of `y` is now, but `cos` didn't print this value to the screen, it returned the value so that `y` could be assigned to that value.

As a diagram, I am going to use the following:

1. functions that *only* display information (and return nothing)

   | `display_function` *(display info here)* |

2. functions that *only* ask for user input (and return it)

   [information] ⟵ | `input_function` *(input information here)* |

3. functions that *only* calculate information, but display *nothing* (and return the result)

   [result] ⟵ | `calculation_function` *(calculate here)* | ⟵ **[information]**

For example, the sin function would be written like:

[result] ← `sin` *(calculate sine of angle here)* ← **[angle]**

Let's rewrite the `interest` program from earlier, in terms of functions. Looking at the code above, we have a couple of parts which input a positive number from the user, and something which calculates the principal given the original princial, rate, and year. A couple of useful functions would then be:

[positive number] ← `input_positive_number` *(display string, input number information here)* ← **[string]**

[new_principal] ← `get_principal` *(calculate new principal here)* ← **[initial_principal, rate, current_year]**

In Python these functions look like

```
def input_positive_number(prompt):

    y=-1
    while y<0:
        y=input(prompt)
        if y<0:
            print 'This value cannot be negative.  Please reenter it.'

    return y

#----------------------------------------

def get_principal(principal_orig,rate,year):

    principal_final=principal_orig*(1.0+rate)**year;

    return principal_final

#----------------------------------------

principal=input_positive_number('What is the initial principal? ')
rate=input_positive_number('What is the annual interest rate? ')

if rate>1:
    print 'Converting percentage rate to decimal rate'
    rate=rate/100.0

number_of_years=input_positive_number('How many years do you want to calculate interes

print 'The original principal is $',principal
print 'The interest rate is ',rate

for current_year in range(1,number_of_years+1):

    new_principal=get_principal(principal,rate,current_year)

    print 'After year ',current_year,': '
```

```
    print '   The interest is $',new_principal-principal
    print '   And the new principal is $',new_principal
```

Let's step through **input_positive_number** function to see what is going on. In the main code there is a line

```
principal=input_positive_number('What is the initial principal? ')
```

This line calls the function **input_positive_number**, and gets to the first line of that function which is:

```
def input_positive_number(prompt):
```

This says "assign the value of `'What is the initial principal?  '` to the variable `prompt`" The variable `prompt` is *local* to this function, which means that it cannot be seen from the outside, and will not conflict with any other variable called `prompt` in any other function or script. For example, if I had done:

```
prompt='hello';
principal=input_positive_number('What is the initial principal? ');
disp(prompt);
```

then the displayed string would be **hello**, because this script doesn't see the internal **prompt** variable in the **input_positive_number** function.

In a function, the *order* of the input arguments is what assigns data to a variable, not the variable names themselves. Variable names are not absolute in this case. For example, the following code...

```
def fun1(x,y):

    print "In fun1:"
    print "  x is ",x
    print "  y is ",y

def fun2(y,x):

    print "In fun2:"
    print "  x is ",x
    print "  y is ",y



a=5
b='hello'

fun1(a,b)
fun2(a,b)
```

... results in the following output:

```
In fun1:
  x is  5
  y is  hello
In fun2:
  x is  hello
  y is  5
```

Input and Output Arguments versus `input` and `print` Commands

Beginning programmers often get confused about the difference between *displaying* a result and *returning* a result. For example, look at the following two functions:

```
def squared_disp(x):
    y=x*x
    print y
```

```
def squared(x):
    y=x*x
    return y
```

In the first case, `squared_disp`, the result of the square is displayed on the screen. The caller of the function might run it like

```
>>> squared_disp(3)
```

and "9" is displayed on the screen. Then what? That's pretty much the extent of the usefulness of this function (which is not particularly useful).

The second case, `squared`, assigns the variable, `y`, to the value of `x*x`, and returns it. From there, the caller of the function can choose to display it, or not, or use it in a further calculation.

```
>>> squared_disp(3)
9
>>> squared(3)
9
>>> z=squared(3)
>>> z+5
14
>>> a=9
>>> b=12
>>> import math
>>> c=math.sqrt(squared(a)+squared(b))
>>> print c
15.0
```

Returned values are much more useful than displayed values.

More Turtle Examples

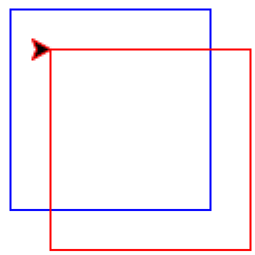
The easiest example of a use of a function is to bundle together a number of commands into a convenient shortcut. Consider the following example to draw a square. Here the function `square` draws a square of a given size. When the command `square(100)` is done, then inside the function the variable `sz` is assigned to 100 and then used throughout the rest of the function.

Listing 2.10: "Drawing Two Squares with Functions"

```
from turtle import *
reset()

def square(sz):
    for side in range(4):
        forward(sz)
        right(90)

```

```
pencolor("blue")
square(100)

penup()
forward(20)
right(90)
forward(20)
left(90)
pendown()

pencolor("red")
square(100)
```

...which draws...



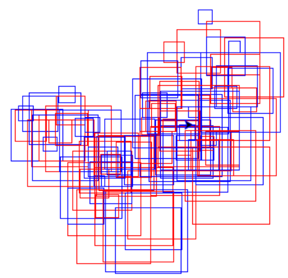
A more complex example is the following, which uses the function, if-then structure, for-loop structure and random numbers to draw an interesting, somewhat random, pattern. The function `randint` returns a random number between the two numbers given. For example, `randint(1,10)` returns a random number between 1 and 10 inclusive. Notice how the alternating red/blue patterns is done. Calling `pencolor()` with nothing inside returns the current color of the pen. We check to see if the current color is blue, and if it is, we set it to red. Otherwise we set it to blue.

Listing 2.11: "Drawing Many Random Squares"

```
from turtle import *
from random import randint

reset()
speed(0)

def square(sz):
    for side in range(4):
        forward(sz)
        right(90)

pencolor("blue")

for i in range(100):

    size=randint(10,100)
    square(size)

    move_over=randint(-30,30)
    move_up=randint(-30,30)

    penup()
```

```
    forward(move_over)
    right(90)
    forward(move_up)
    left(90)
    pendown()

    if pencolor()=='blue':
        pencolor('red')
    else:
        pencolor('blue')
```

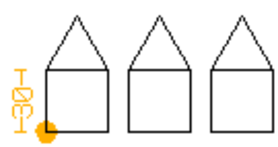
...which draws...



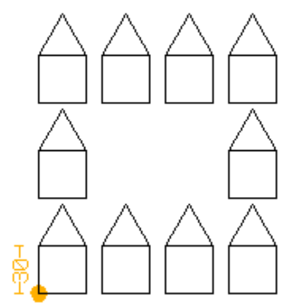
Exercise 2.6

Write programs to draw the following patterns:

Pattern 1



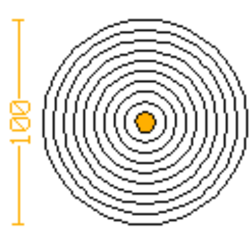
Pattern 2



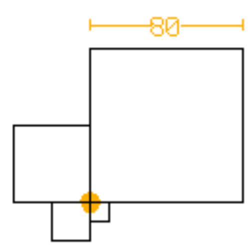
Exercise 2.7

Write programs to draw the following patterns:

Pattern 3



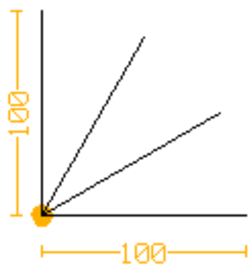
Pattern 4



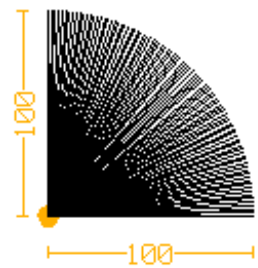
Exercise 2.8

Write programs to draw the following patterns:

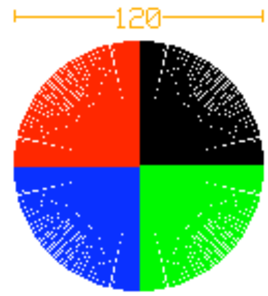
Pattern 5



Pattern 6



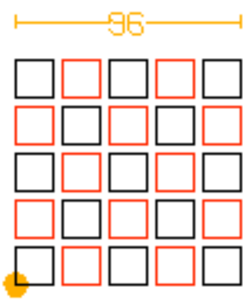
Pattern 7



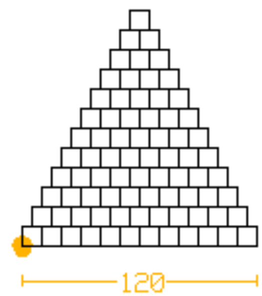
Exercise 2.9

Write programs to draw the following patterns:

Pattern 8



Pattern 9



Exercise 2.10

Make a function to draw a polygon. It should be given the number of sides, and the length of the side. For example, if the number of sides is, say, 3 then it should draw a triangle. If it's 4 then a square, 5 a pentagon, etc...

Local Variables

Variables assigned within a function have no connection to any variables outside of the function, even those of the same name. The variable is considered *local* to the function, and is destroyed when the function ends. The only way to pass information out of a function is through the `return` statement.

Input arguments function just like local variables: changes in their value do not affect anything outside of the function itself. Some examples will help elucidate the properties of local variables. Consider the following functions:

```
def localvars_second():

    a=10
    a=a+1

    print "second a is ",a


```

```
def localvars_third(b):

    b=b+1
    a=2*b

    print "third a is ",a


def localvars_fourth(c,e):

    print "beginning of fourth c is ",c
    print "beginning of fourth e is ",e

    a=c*c
    d=a+1
    d=d-2

    c=0
    e=-1


    print "fourth a is ",a
    print "fourth c is ",c
    print "fourth e is ",e
    print "fourth d is ",d

    return d


def localvars_first():

    a=5
    d=1

    localvars_second()
    print "first a is ",a

    localvars_third(a)
    print "first a is ",a

    y=localvars_fourth(a,2*a)
    print "first a is ",a
    print "first d is ",d
    print "first y is ",y

#-------------------------------------------------
# call it
localvars_first()
```

What will be printed when we run `localvars_first()`? Here is the resulting output:

```
>>> localvars_first()
second a is  11
first a is  5
third a is  12
first a is  5
```

```
beginning of fourth c is  5
beginning of fourth e is  10
fourth a is  25
fourth c is  0
fourth e is  -1
fourth d is  24
first a is  5
first d is  1
first y is  24
```

The logic of the output is as follows. The function `localvars_first` sets the value of `a` to be 5, and then the value of `d` to be 1. It then calls `localvars_second`, which sets *its own* variable `a` to 10, and then adds one to that, yielding 11. This does *not* affect the value of `a` in `localvars_first`.

The function `localvars_first` then calls `localvars_third`, passing it the value of `a`, which is still 5. This value is copied to the variable `b` in `localvars_third`, because we called the first input argument `b`. Now `b` will act like a local variable, and any changes to it will not be seen outside of the `localvars_third` function. Again, `localvars_third` sets *its own* variable `a`.

The function `localvars_first` then calls `localvars_fourth`, passing it two values, one of which is the value of `a`, which is still 5, and the other is the value of `2*a`, which is 10. These are copied to the *local* variables in `localvars_fourth`, `c` and `e` respectively. Changes to these variables, or local versions of `a`, do not affect the value of `a` in `localvars_first`. The value of `d`, which is returned from `localvars_fourth`, gets copied to `y` in `localvars_first`.

Some Useful Python Functions

In this section we summarize a few useful Python functions.

- `math.floor` and `math.ceil`: These functions take a real number and return another real number, closest to an integer. `floor` drops anything after the decimal, rounding down always. `ceil` rounds up always.

```
>>> import math
>>> a=6.3425
>>> b=2.786
>> math.floor(a)
6.0
>>> math.ceil(a)
7.0
```

- `random.random`: This function returns a random number between 0 and 1, not including 0 or 1.

```
>>> import random
>>> random.random()
0.98301863463418293
>>> random.random()
0.64299904798504881
```

If you want to make a random number, say, from 1 to 10, then you can use the `random.randint` function:

```
>>> import random
>>> random.randint(1,10)
2
>>> random.randint(1,10)
7
>>> random.randint(1,10)
6
```

Exercise 2.11

Write the following functions to work, as stated. The name of the function should give you clue about it's general behavior.

- ```python
def square(x):
    """Returns the square of x, where x is a number.

    >>> square(4)
    16
    >>> square(12)
    144

    """
```

```python
def ispositive(x):
    """Returns True if x is a positive number

    >>> ispositive(4)
    True
    >>> ispositive(0)
    False
    >>> ispositive(-1)
    False
    """
```
```

Exercise 2.12

Write the following functions to work, as stated. The name of the function should give you clue about it's general behavior.

```python
def threshold(x,theta=0):
    """Returns x if x is greater than theta, and theta otherwise.

    >>> threshold(5)
    5
    >>> threshold(-4)
    0
    >>> threshold(5,10)
    10
    >>> threshold(12,10)
    12

    """
```

```python
def threshold2(x,theta1=-1,theta2=1):
    """Returns x if x is between theta1 and theta2.

    If theta1<theta2 then
        returns theta1 if x is less than theta1,
        returns theta2 if x is greater than theta2.
    if theta1>theta2, then the roles of theta1 and theta2
    are swapped

    >>> threshold2(5)
    1
    >>> threshold2(-2)
    -1
    >>> threshold2(0.5)
    0.5
    >>> threshold2(-0.5,0,1)
    0
    >>> threshold2(0.5,-1,0)
    0
    >>> threshold2(-0.5,0,-1)  # should work
    -0.5

    """
```

Exercise 2.13

1. Modify `acey_ducey` written earlier to be broken up into functions. You should have:
   - `print_rules`: prints the rules of the game
   - `get_card`: returns a random card
   - `print_card`: prints the value of a card
   - `get_valid__bet`: given two cards as input arguments, and the amount of money, return a valid user-input bet
   - `is_between`: given three cards as input arguments, returns 1 if the third card is in between the other two and return 0 otherwise
   - `acey_ducey2`: the main function which runs the game
2. Write a version of `acey_ducey`, calling it `acey_ducey3`, which uses a different version of `get_bet`, say `get_computer_bet`, where the computer gives a reasonable bet given the two cards as input arguments.

Chapter 3

Program Design

3.1 Guidelines

You now know enough Python syntax to do almost any program. There are other parts of Python syntax that we will cover later, but it is important to pause and consider how programs are designed. A programmer *never* starts by just sitting at a keyboard and typing. There are several key steps before any code is written, or the programmer will waste hours, even days, with false starts and debugging issues.

A programmer starts with a problem, and usually some idea of how to solve it. The task of the programmer is to translate this idea into a step-by-step set of instructions to solve the problem. The basic approach is the following

- Break up the problem into smaller pieces, which can be tested individually. This is usually done by writing the program like a recipe with parts of the problem in English. You may have things like “do the following 5 times” or “calculate this value from the other values given”. These will be fleshed out later in the process. You should have a full recipe for the problem, written out on paper, well before you start to type.
- Attack each smaller piece as a separate program, breaking into smaller pieces as necessary.
- You should find that your more refined recipe consists of a set of functions, each of which is small and does a very specific thing.
- Desk-check the recipe to see that it really works. This means stepping through your recipe, on paper, as if you were a computer, not taking anything for granted, and seeing that it works. You should also desk-check what your functions do when given improper input. The desk-check is a very important step, and is often skipped by beginning programmers who are in a rush to get to the keyboard. It also is a step that, when skipped, wastes more time than missing almost any other step!
- Identify what information each function needs to get from the main program. These will be your input arguments for that function. Identify what information each function needs to return to the main program. These will be its output arguments.
- You then write each function in proper code, and test it *individually*. This is the important thing: **do not trust that a function works without testing it**. Put in bad values, and make sure it behaves well. Test it over the entire range of possible input values, especially values like zero or the maximum valid value, to make sure the function is robust.
- When you start putting the functions together to match your recipe, do it one function at a time, and test it. If you type the whole recipe out at once, and it doesn’t work, then you don’t where it is failing, and it will take you a long time to figure out where it is failing. Testing after the addition of *any* amount of code saves you time in the long run.

Here are some guidelines that help with programming and debugging.

- Other than your main function, all other functions should only return values and *not* print things to the screen unless that is their *only* job. Let the main function print out the values if it needs to, or not if it doesn't need to. A function is much less useful if it prints values to the screen itself.

- 90% of debugging is *preventative*. If you are careful, test every bit of code you add (no matter how small or insignificant) *at the time you add it*, and follow the guidelines below you will avoid many pitfalls.

- Name variables consistently. If you use use variables like `interestrate`, `interest_rate`, `InterestRate`, etc. then make *all* variables like that. You don't want to use `InterestRate` in one place and `mortgage_rate` somewhere else. It gets confusing!

- If you get an error, determine what the error is before you try to change your code to fix it. Although often terse and difficult to read, the error messages do tell you what is wrong (although it takes some practice to interpret them).

- Test each function separately, and over a wide range of possible input values. You can only trust a program if each function is working perfectly.

- Do not try to write the entire program before testing. This becomes a nightmare fast! Break things into pieces, and test each piece, no matter how trivial it seems.

- When you really get stuck, have someone else look at your code. Seeing with different eyes often allows someone to see errors that you've been staring at (and missing) for an hour.

- When trying to determine why something is going wrong, put in a lot of `print` statements to confirm that the variables have the values you think they do. Do this even in cases where it is obvious that the variable is correct, because sometimes the obvious is not true.

- And finally, remember the golden rule of debugging: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong.

3.2 Tank Wars: An Extended Example

As an example, I will step through the development and coding process for a game of tank wars. The rules of the game are as follows:

- The game is played between two people, each of which has a tank.

- The two tanks are 1000 m apart, and are facing each other.

- On a given turn, each sets the elevation angle of their cannon and the initial speed of the shot to be fired.

- Both tanks fire simultaneously. The distance (in meters) that the shot travels before hitting the ground is given by physics:

  $$\text{distance} = \frac{(\text{initial shot speed})^2 \times \sin(2 \cdot \text{angle}) + 2(\text{wind speed})(\text{initial shot speed})\sin(\text{angle})}{10}$$

  where the wind is traveling from left to right.

- If a shot comes within 10 meters of any tank, then that tank is destroyed.

- If a tank is destroyed, then that player loses. If both tanks are destroyed in the same turn, then it is a stalemate.

- The wind speed is a constant set in the program, and displayed at the beginning of the game.

Writing the Recipe

So how do we begin to write such a program? We first write a recipe for the program, mostly in English. Something like:

- Set Wind Speed
- While No One Has Won
  - Get Each Player's Angle and Speed
  - Get Where Each Shot Landed
  - Display Where Each Shot Landed
  - Determine Who Has Been Destroyed, If Anyone
  - Determine Who Has Won, If Anyone
- Display Who Won

Some of these parts will translate easily into Python code, while others will take some steps. Now let's go to the next stage of refinement. Lines like Set Wind Speed can be translated directly as `wind_speed=5;`

The while-loop will look something like

```
no_one_has_won=True
while (no_one_has_won):

    STUFF HERE.  WHEN SOMEONE WINS, OR A STALEMATE, THEN
    WE WILL WRITE no_one_has_won=False
```

This is a very common structure for a while-loop, where you have a variable which starts off true, and then is set to false when you want to not continue repeating the while-loop.

Designing the Functions

The line Get Each Player's Angle and Speed should be a set of two statements Get Player 1's Angle and Speed and Get Player 2's Angle and Speed. If we use a single function for this, we would want something simply like `get_angle_and_speed`. What information does this function need to be given? Just the player's number, 1 or 2. What information does this function return? Two numbers: the angle and the speed. So we should have something like

[angle,speed] ⟵ | `get_angle_and_speed` *(display player number, input angle and speed here)* | ⟵ **[player number]**

```
def get_angle_and_speed(player_number):

    print 'Player ',player_number
    angle=input('  Enter your Angle of Elevation: ')
    speed=input('  Enter your Angle of Speed: ')

    return angle,speed
```

Right now this doesn't check to see if the velocity is above zero, or the elevation is between 0 and 90 degrees, but this can be added easily.

```
def get_angle_and_speed(player_number):

    print 'Player ',player_number
```

```
    angle=input('  Enter your Angle of Elevation: ')

    if (angle<0) or (angle>90): # illegal angles
        raise ValueError,"Illegal Angle Given"

    speed=input('  Enter your Angle of Speed: ')

    if speed<0:
        raise ValueError,"Illegal Speed Given"


    return angle,speed
```

Now we test this function, using several values of the player number, and also testing the illegal values of angles and speeds.

```
>>> execfile('tankwars.py')
>>> angle1,speed1=get_angle_and_speed(1)
Player 1
  Enter your Angle of Elevation: 45
  Enter your Angle of Speed: 100
>>> angle1
45
>>> speed1
100
>>> angle2,speed2=get_angle_and_speed(2)
Player 2
  Enter your Angle of Elevation: 145
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "tankwars.py", line 7, in get_angle_and_speed
    raise ValueError,"Illegal Angle Given"
ValueError: Illegal Angle Given
>>> angle2,speed2=get_angle_and_speed(2)
Player 2
  Enter your Angle of Elevation: 45
  Enter your Angle of Speed: 50
>>> angle2,speed2=get_angle_and_speed(2)
Player 2
  Enter your Angle of Elevation: 45
  Enter your Angle of Speed: -50
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "tankwars.py", line 12, in get_angle_and_speed
    raise ValueError,"Illegal Speed Given"
ValueError: Illegal Speed Given
```

Continuing our program, we need a function to Get Where Each Shot Landed. Again, we split this into Get Where Player 1's Shot Landed and Get Where Player 2's Shot Landed, again as a single function. Since this function does not need to print anything, it doesn't need to know who the player is. It just needs the angle of elevation, the shot speed, and the wind speed. It will then return the distance, so it should look like

[distance] ← `get_shot_distance` *(calculate distance here)* ← **[angle,shot speed,wind speed]**

```
def get_shot_distance(angle,shot_speed,wind_speed):
```

Now, in Python, all trigonometric functions are given in *radians*, not *degrees*. We will have to translate from degrees to radians to use the equation for the distance. This calls for another function, that is given an angle in degrees and returns the value in radians.

[angle in radians] $\longleftarrow$ `radians` *(calculate radians here)* $\longleftarrow$ **[angle in degrees]**

```
def radians(d):
    r=d*3.1415926535897932/180
    return r
```

Test this function knowing that 0 degrees is 0 radians, 180 degrees is π radians, and 360 degrees is 2π radians.

```
>>> radians(0)
0.0
>>> radians(180)
3.1415926535897931
>>> radians(360)
6.2831853071795862
```

Now we are ready to make our `get_shot_distance` function.

```
function distance=get_shot_distance(angle,shot_speed,wind_speed)

angle=radians(angle);
distance=(shot_speed^2*sin(2*angle)+2*wind_speed*shot_speed*sin(angle))/10;

endfunction
```

To test this function, we should desk-check a few values. We can also do a couple of cases like shooting straight up with no wind, shooting straight up with a wind, shooting at an angle of 0, or a velocity of 0.

```
>>> get_shot_distance(90,20,0) # really small value (effectively zero)
4.898425415289509e-15
>>> get_shot_distance(90,20,10) # wind pushes the shot forward
40.000000000000007
>>> get_shot_distance(90,20,-10)  # wind pushes the shot backward
-39.999999999999993
>>> get_shot_distance(0,20,10)  # hits the ground immediately
0.0
>>> get_shot_distance(45,0,10) # hits the ground immediately
0.0
```

Our program now looks like

```
wind_speed=5

no_one_has_won=True
while no_one_has_won:

    angle1,speed1=get_angle_and_speed(1)
    angle2,speed2=get_angle_and_speed(2)

```

```
    distance1=get_shot_distance(angle1,speed1,wind_speed)
    distance2=get_shot_distance(angle2,speed2,-wind_speed)

    MORE STUFF
```

Notice that we used `-wind_speed` for player 2, because it experiences the opposite wind pattern. Now we have to display these distances, using `print` statements, and determine a winner. The function to determine a winner should take the two distance values, and return one of 4 values. 0: no one hit anything, 1: player one won, 2: player two won, 3: both destroyed.

[winning player (0 for no win, 3 for tie)] ⟵ `get_winning_player` *(calculate winner here)* ⟵ **[distance1,distance2]**

```
def get_winning_player(distance1,distance2):

    tank_1_hit=(distance2>=990) and (distance2<=1010)
    tank_2_hit=(distance1>=990) and (distance1<=1010)

    if tank_1_hit and tank_2_hit:
        winning_player=3  # stalemate
    elif tank_1_hit: # only tank 1 hit
        winning_player=2
    elif tank_2_hit: # only tank 2 hit
        winning_player=1
    else: # no tanks hit
        winning_player=0

    return winning_player
```

Again we test with a range of values, especially those on the extremes (990 and 1010).

```
>>> get_winning_player(500,500)
0
>>> get_winning_player(1001,500)
1
>>> get_winning_player(1010,500)
1
>>> get_winning_player(1011,500)
0
>>> get_winning_player(1011,990)
2
>>> get_winning_player(1010,990)
3
```

The Final Program

Adding a few display statements, and an if-statement to determine what message to print, we have a complete program.

```
wind_speed=5

print 'The wind speed is ',wind_speed

no_one_has_won=True

```

```
while no_one_has_won:

    angle1,speed1=get_angle_and_speed(1)
    angle2,speed2=get_angle_and_speed(2)

    distance1=get_shot_distance(angle1,speed1,wind_speed)
    distance2=get_shot_distance(angle2,speed2,-wind_speed)

    print 'Player 1 Shot a Distance of ',distance1
    print 'Player 2 Shot a Distance of ',distance2

    winning_player=get_winning_player(distance1,distance2)

    if (winning_player>0):
        no_one_has_won=False

if winning_player==1:
    print 'Player 1 Won!'
elif winning_player==2:
    print 'Player 2 Won!'
else:
    print 'Stalemate.'
```

Notice that the functions and variables are named in such a way that the final program looks a lot like the recipe. A sample run is as follows:

```
>> execfile('tankwars.py')
The wind speed is  5
Player  1
Enter your Angle of Elevation: 45
Enter your Angle of Speed: 40
Player  2
Enter your Angle of Elevation: 30
Enter your Angle of Speed: 100
Player 1 Shot a Distance of  188.3
Player 2 Shot a Distance of  816
Player  1
Enter your Angle of Elevation: 45
Enter your Angle of Speed: 150
Player  2
Enter your Angle of Elevation: 30
Enter your Angle of Speed: 210
Player 1 Shot a Distance of  2356
Player 2 Shot a Distance of  3714
Player  1
Enter your Angle of Elevation: 45
Enter your Angle of Speed: 110
Player  2
Enter your Angle of Elevation: 30
Enter your Angle of Speed: 150
Player 1 Shot a Distance of  1288
Player 2 Shot a Distance of  1874
Player  1
Enter your Angle of Elevation: 45
Enter your Angle of Speed: 97
Player  2
```

```
Enter your Angle of Elevation: 30
Enter your Angle of Speed: 115
Player 1 Shot a Distance of  1009
Player 2 Shot a Distance of  1088
Player 1 Won!
```

Exercise 3.14

Create the following computer game: The computer randomly selects an integer between 1 and 100. The user has to guess the number in the fewest number of tries. After each guess, the computer tells you whether the guess is too high or too low. At the end of the game print out the number of guesses it took. After each game, the user has the option of continuing with another game. **Make sure to write a recipe for the program before you write any Python code, and include it with your program.**

Exercise 3.15

Make the same game, but this time have the human pick the number, and the computer guesses. The computer should need no more than 8 guesses to win.

Chapter 4

Program Structure - Part II

Up until this point, everything we have presented is true for practically any programming language. They all have a branching structure, and a looping structure. They all deal with boolean variables, and have some form of function or subroutine structure to break up problems into smaller pieces. Now we add lists and dictionaries to the mix, and really add a lot of power to our programs.

4.1 Lists

Python supports lists as a basic data structure. For example, you can do

```
>>> a=[-4,4,10,-2,20]
>>> a
[-4, 4, 10, -2, 20]
>>> a[3]
-2
>>> a[0]
-4
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Notice that you can access the elements of a list like `a[2]`, and that the elements are numbered starting with **0**, not 1. If you try to access beyond the length of a list, then an error results.

When using the multiply operator, `*`, the list gets duplicated. For example,

```
>>> a=[1]*5
>>> a
[1, 1, 1, 1, 1]
```

The length of a list is given by the function `len`. This lets you cycle through the values of a list easily.

```
a=[-4,4,10,-2,20]
for i in range(N):
    if a[i]>0:
        print "The element number ",i," is greater than zero"
```

which results in

```
The element number  1  is greater than zero
The element number  2  is greater than zero
```

```
The element number 4 is greater than zero
```

A Warning about Copying Lists

In the way that Python works, if you do:

```
>>> a=[1,2,3,4,5]
>>> b=a
```

Then `b` is not a copy of `a`, but the **same list** as `a`. Modifying `b` also modifies `a`, for example

```
>>> a=[1,2,3,4,5]
>>> b=a
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>> b[2]=100
>>> b
[1, 2, 100, 4, 5]
>>> a
[1, 2, 100, 4, 5]
```

To avoid this, do the following

```
>>> a=[1,2,3,4,5]
>>> b=a[:]  # make a copy of a
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>> b[2]=100
>>> b
[1, 2, 100, 4, 5]
>>> a
[1, 2, 3, 4, 5]
```

4.2 Extended Example: Tank Wars with N Players

If we wanted to extend our previous tank wars example from 2 players to any amount of players, we would have to make some organizational changes to reflect multiple players. In this version, the wind speed will be set randomly each turn, not just at the beginning of the game.

- Get Number of Players
- Get The Tank Positions
- While No One Has Won
    - Set Wind Speed
    - Display The Tank Positions and Wind Speed
    - For Each Surviving Player...
        - Get Their Angle and Speed

    - Get Where Each Shot Landed
  - Display Where Each Shot Landed
  - Determine Who Has Been Destroyed, If Anyone
  - Determine Who Has Won, If Anyone

- Display Who Won

Almost all of the structure will be the same as the 2 player game. To get the number of players, we can use the same structure as we used before for getting values typed in by the user

```
def get_number_of_players():
    N=input('How many players? ')

    if N<=1:
        raise ValueError,'Illegal number of players'

    return N
```

To get the tank positions, we choose random numbers from 0 to 1000. We need `N` of them, so we want to make a list of length `N` (using the `*` operator), and then filling in the values using `random.random()` to make a random values containing values from 0 to 1. We can then multiply by 1000 to get the positions.

```
def get_tank_positions(N):

    pos=[0]*N   # make a list of length N, with zeros

    # fill in all the values with random numbers
    for i in range(N):
        pos[i]=random.random()*1000.0

    return pos
```

Next we must write out how we do For Each Surviving Player. Somehow we need to keep track of which players are dead, and only ask for input from non-dead players. To do this, let's make another list, called `isdead`, which is `False` for all alive players and `True` for all dead players. Then

- While No One Has Won
  - Set Wind Speed
  - Display The Tank Positions and Wind Speed
  - For Each Surviving Player...
    - Get Their Angle and Speed
    - Get Where Each Shot Landed
  - Display Where Each Shot Landed
  - Determine Who Has Been Destroyed, If Anyone
  - Determine Who Has Won, If Anyone
- Display Who Won

Changes to

- Initialize `isdead` List to all Zeros
- While No One Has Won

    - Set Wind Speed
    - Display The Tank Positions and Wind Speed
    - For Each Player...
      - If the Player is Not Dead...
        - Get Their Angle and Speed
        - Get Where Each Shot Landed
    - Display Where Each Shot Landed
    - Determine Who Has Been Destroyed, If Anyone
    - For all of those destroyed, set `isdead` to `True`
    - Determine Who Has Won, If Anyone

- Display Who Won

The statement Get Where Each Shot Landed in this case will be translated to Make a List of Positions of Where Each Player's Shot Landed. This is the sum of the previously written function `get_shot_distance` and the current tank position. The inner-most loop now becomes

```
    for player in range(N):

        # get all of the angles and speeds

        if not isdead[player]:
            [angle,speed]=get_angle_and_speed(player)

            distance=get_shot_distance(angle,speed,wind_speed)

            # make a list with all of the places the shots landed
            shot_pos[player]=tank_pos[player]+distance
```

We have to update the `get_angle_and_speed` function to accept angles from 0 to 180, instead of from 0 to 90, so the tanks can fire both to the right and to the left. In that function

```
    if (angle<0) or (angle>90): # illegal angles
```

becomes

```
    if (angle<0) or (angle>180): # illegal angles
```

To Determine Who Has Been Destroyed, If Anyone, we must go through all players (the same type of loop), and check to see if any of the shots were within range. We should make a function, called `isdestroyed`, to return `True` if the tank is destroyed. What information does this function need? It certainly needs to know which tank we are testing to see if it is destroyed, the positions of the tanks, the positions of the shots, and which tanks are dead. Thus, it's syntax should be something like `isdestroyed(player,tank_pos,shot_pos,isdead)`. A dead tank should return `False`.

```
def isdestroyed(current_player,tank_pos,shot_pos,isdead):

    if isdead[current_player]:
        return False # a dead one cannot be destroyed

    N=len(tank_pos)

    for player in range(N):  # players numbered from 0 to N-1
        if not isdead[player]:  # did the player's shot hit the current player?
```

```
            if abs(shot_pos[player]-tank_pos[current_player])<10: # a hit
                return True


    # if you've gotten this far past the loop, then you're not destroyed

    return False
```

Notice how we obtained the value for N in the function. Since we can't use N without assigning it a value, we could have passed the value of N as a parameter. Instead (just to make one less parameter) we determined N from the properties of the other parameters, namely the length of the tank position list. This saves us one more parameter to pass, and makes the code a bit cleaner.

To print out the tank positions, what we need to do is to go through all of the players, print one message for the ones which are dead (`isdead(player)`), and another for those that are still surviving.

```
def print_tank_positions(pos,isdead):
    N=len(pos)

    for player in range(N):
        if isdead[player]:
            print 'Player ',player,' is dead.'
        else:
            print 'Player ',player,' is at position ',pos[player],'.'
```

A very similar function for printing the shot positions, except we don't have to write anything for those dead tanks, only the ones that are not dead.

```
def print_shot_positions(pos,isdead):

    N=len(pos)

    for player in range(N):
        if not isdead[player]:
            print 'Shot for player ',player,' landed at position ',pos[player],'.'
```

How do we determine if there is a winner? Logically, it means that there is only one survivor. How do we determine this from the variables we have? If we could count the number of `True` values in the `isdead` list, that would be the number of tanks killed. N minus this number is the number of survivors. We may want to also keep track of *which* tank is alive, if any.

```
    # find out who has been destroyed
    dead_count=0
    last_alive=-1  # keep track of a live one
    for player in range(N):
        if isdestroyed(player,tank_pos,shot_pos,isdead):
            print 'Player ',player,' has been destroyed.'
            isdead[player]=True

        if isdead[player]:
            dead_count=dead_count+1
        else:
            last_alive=player

    number_alive=N-dead_count

    if number_alive<2:
```

```
        no_one_has_won=False  # break out of loop
```

Finally, we have all of the pieces together, and the complete program is

```
import math
import random

def get_angle_and_speed(player_number):

    print 'Player ',player_number
    angle=input('  Enter your Angle of Elevation: ')

    if (angle<0) or (angle>180): # illegal angles
        raise ValueError,"Illegal Angle Given"

    speed=input('  Enter your Speed: ')

    if speed<0:
        raise ValueError,"Illegal Speed Given"


    return angle,speed

def get_number_of_players():
    N=input('How many players? ')

    if N<=1:
        raise ValueError,'Illegal number of players'

    return N

def get_tank_positions(N):

    pos=[0]*N   # make a list of length N, with zeros

    # fill in all the values with random numbers
    for i in range(N):
        pos[i]=random.random()*1000.0

    return pos

def isdestroyed(current_player,tank_pos,shot_pos,isdead):

    if isdead[current_player]:
        return False # a dead one cannot be destroyed

    N=len(tank_pos)

    for player in range(N):  # players numbered from 0 to N-1
        if not isdead[player]:  # did the player's shot hit the current player?
            if abs(shot_pos[player]-tank_pos[current_player])<10: # a hit
              return True


    # if you've gotten this far past the loop, then you're not destroyed

```

```
    return False

def print_shot_positions(pos,isdead):

    N=len(pos)

    for player in range(N):
        if not isdead[player]:
            print 'Shot for player ',player,' landed at position ',pos[player],'.'

def print_tank_positions(pos,isdead):
    N=len(pos)

    for player in range(N):
        if isdead[player]:
            print 'Player ',player,' is dead.'
        else:
            print 'Player ',player,' is at position ',pos[player],'.'

def get_angle_and_speed(player_number):

    print 'Player ',player_number
    angle=input('  Enter your Angle of Elevation: ')

    if (angle<0) or (angle>90): # illegal angles
        raise ValueError,"Illegal Angle Given"

    speed=input('  Enter your Angle of Speed: ')

    if speed<0:
        raise ValueError,"Illegal Speed Given"


    return angle,speed


def radians(d):
    r=d*3.1415926535897932/180
    return r

def get_shot_distance(angle,shot_speed,wind_speed):

    angle=radians(angle)
    distance=(shot_speed**2*math.sin(2.0*angle)+
              2.0*wind_speed*shot_speed*math.sin(angle))/10.0;


    return distance

#==================================================================


N=get_number_of_players()

tank_pos=get_tank_positions(N)
```

```

# no one starts out dead (0 is the same as false)
isdead=[False]*N

# shot positions start off as zero
shot_pos=[0]*N

no_one_has_won=True
while no_one_has_won:

    wind_speed=(random.random()*20)-10;  #random speed from -10 to 10
    print 'The wind speed is ',wind_speed
    print_tank_positions(tank_pos,isdead)

    for player in range(N):

        # get all of the angles and speeds

        if not isdead[player]:
            [angle,speed]=get_angle_and_speed(player)

            distance=get_shot_distance(angle,speed,wind_speed)

            # make a vector with all of the places the shots landed
            shot_pos[player]=tank_pos[player]+distance

    print_shot_positions(shot_pos,isdead)

    # find out who has been destroyed
    dead_count=0
    last_alive=-1
    for player in range(N):
        if isdestroyed(player,tank_pos,shot_pos,isdead):
            print 'Player ',player,' has been destroyed.'
            isdead[player]=True

        if isdead[player]:
            dead_count=dead_count+1
        else:
            last_alive=player

    number_alive=N-dead_count

    if number_alive<2:
        no_one_has_won=False  # break out of loop



if number_alive==0:
    print 'Everyone is dead. Stalemate.'
else:
    print 'Player ',last_alive,' has won!'
```

Run like

```
>> execfile('tankwarsN.py')
```

```
How many players?4
The wind speed is  -4.88
Player  1  is at position  392 .
Player  2  is at position  994.6 .
Player  3  is at position  824.2 .
Player  4  is at position  203.9 .
Player  1
Enter your Angle of Elevation: 35
Enter your Angle of Speed: 25
Player  2
Enter your Angle of Elevation: 50
Enter your Angle of Speed: 100
Player  3
Enter your Angle of Elevation: 45
Enter your Angle of Speed: 35
Player  4
Enter your Angle of Elevation: 67
Enter your Angle of Speed: 100
Shot for player  1  landed at position  436.8 .
Shot for player  2  landed at position  1905 .
Shot for player  3  landed at position  922.5 .
Shot for player  4  landed at position  833.4 .
Player  3  has been destroyed.
The wind speed is  -5.314
Player  1  is at position  392 .
Player  2  is at position  994.6 .
Player  3  is dead.
Player  4  is at position  203.9 .
Player  1
Enter your Angle of Elevation:
```

Exercise 4.16

Write the following functions to work, as stated. The name of the function should give you clue about it's general behavior.

-
```
def sumlist(mylist):
    """Returns the sum of the values in the list mylist.

    >>> sumlist([1,2,3])
    6
    >>> sumlist([])
    0
    >>> sumlist([1.1,2.2,3.3,4.4])
    11.0


    """
```

```
def squarelist(mylist):
    """Returns a list the same size as mylist, with all the elements squared

    >>> squarelist([1,2,3])
    [1, 4, 9]
    >>> squarelist([5,1,2,3])
    [25, 1, 4, 9]
    >>> squarelist([])
    []

    """
```

Exercise 4.17

An extended version of the Nim game goes as follows. You have 4 piles of objects, initially with 7, 5, 3, and 1 objects, respectively. You take turns with an opponent picking up objects, with the following rules:

- you have to pick up *at least 1 object*
- you can only pick from one pile in a turn
- you can pick up as many objects as you like from the pile
- whoever takes the last object *loses*

1. write a recipe for this game, complete with function diagrams as above.
2. write the game allowing for human vs human and human vs computer play

Make sure to use a list for the pile amounts, like
`v=[7, 5, 3, 1];`
and
`v[chosen_pile]=v[chosen_pile]-objects_picked_up`

4.3 Extended Example: Maze

In this section we write a program to generate a random maze. We will use Aldous-Broder's algorithm, which goes as follows:

> Start with a maze grid with all walls up (a bunch of individual, walled cells). Pick a point, and move to a neighboring cell at random. If the new cell has all of its walls still up, then knock the walls down between the previous cell and the new cell. Keep moving to neighboring cells until all cells have been visited.

We will start with the recipe, and then break each piece down into pieces.

- Make Maze
- Draw Maze

then we expand those pieces

- Make Maze
  - Initialize Maze
  - Pick a point
  - Move to a neighboring cell at random
  - If the new cell has all of its walls still up...
    - Knock the walls down between the previous cell and the new cell
  - Keep moving to neighboring cells until all cells have been visited. → **this turns into a while loop**
- Draw Maze
  - For all of the rows...
    - Draw one row of the maze

then we structure the while-loop

- Make Maze
  - Initialize Maze
  - Pick a cell
  - Count this cell as visited
  - While we still have cells to visit...
    - Move to a neighboring cell at random
    - If the new cell has all of its walls still up...
      - Knock the walls down between the previous cell and the new cell
      - Count this new cell as visited
- Draw Maze
  - For all of the rows...
    - Draw one row of the maze

At this point we have to make some decisions about how to represent the maze itself. Pretty much we need to have a number of rows and columns for the maze, and know which walls are up for the particular cell. For example, say we have the following 3x3 maze, and a summary of the information to describe this maze:

```
+—+—+—+
|     |
+—+ +—+
|     |
+—+ + +
|   | |
+—+—+—+
```

The information in this maze is the following:

Index	Row	Column	N	E	W	S
1	1	1	y	n	y	y
2	2	1	y	n	y	y
3	3	1	y	n	y	y
4	1	2	y	n	n	n
5	2	2	n	n	n	n
6	3	2	n	y	n	y
7	1	3	y	y	n	y
8	2	3	y	y	n	n
9	3	3	n	y	y	y

We could have 4 separate lists (or lists or lists, to get 2-D structure) representing north, south, east, and west walls, or we can be a bit more clever and do it in one. If we assign the following values:

```
M =

   11    8   13
   11    0   12
   11    5    7
```

Here I have set the following values:

$$\begin{aligned} N &= 8 \\ E &= 4 \\ W &= 2 \\ S &= 1 \end{aligned}$$

and simply added them up. In that way, all of the possible combinations of walls are specified by a single number from 0 (no walls) to 15 (all walls = 8+4+2+1). We will want a conversion between this number format to a length 4 list specifying the walls in North, East, West, and South directions (I'll refer to this length-4 list format as the NEWS format). To go from NEWS format to number format we just simply add up the values of N, E, W, and S for every element of the NEWS list which is 1.

```python
def news2num(NEWS):
    # convert a north,east,west,south list to a single number wall value
    North=8; East=4; West=2; South=1;

    num=NEWS[0]*North+NEWS[1]*East+NEWS[2]*West+NEWS[3]*South;

    return num
```

To go the other way we can do

```python
def num2news(num):
    # convert single number wall values to a north,east,west,south list

```

```
    North=8; East=4; West=2; South=1;

    NEWS=[0, 0, 0, 0]

    if (num>=8): # North
        NEWS[0]=1
        num=num-8


    if (num>=4): # East
        NEWS[1]=1
        num=num-4

    if (num>=2): # West
        NEWS[2]=1
        num=num-2

    if (num>=1): # South
        NEWS[3]=1
        num=num-1

    return NEWS
```

More to do here

The full source for the maze program is the following:

```
from turtle import *
import random

def num2news(num):
    # convert single number wall values to a north,east,west,south list

    North=8; East=4; West=2; South=1;

    NEWS=[0, 0, 0, 0]

    if (num>=8): # North
        NEWS[0]=1
        num=num-8


    if (num>=4): # East
        NEWS[1]=1
        num=num-4

    if (num>=2): # West
        NEWS[2]=1
        num=num-2

    if (num>=1): # South
        NEWS[3]=1
        num=num-1

    return NEWS
```

```

def news2num(NEWS):
    # convert a north,east,west,south list to a single number wall value
    North=8; East=4; West=2; South=1;

    num=NEWS[0]*North+NEWS[1]*East+NEWS[2]*West+NEWS[3]*South;

    return num

def idx2rc(idx,M,R,C):
    # convert from index representation to row and column

    r=[i%R for i in idx]
    c=[i//R for i in idx]


    return r,c

def rc2idx(rv,cv,M,R,C):
    #convert from row and column representation to an index

    idx=[]
    for r,c in zip(rv,cv):
        idx.append(r+c*R)


    return idx

def get_direction(idx1,idx2,M,R,C):
    # Given two neighboring indices, returns both the direction from first
    # to the second and the second to the first

    North=8; East=4; West=2; South=1;

    r1,c1=idx2rc(idx1,M,R,C)
    r1=r1[0]
    c1=c1[0]

    r2,c2=idx2rc(idx2,M,R,C)
    r2=r2[0]
    c2=c2[0]

    if  ((r1-r2)==0) and ((c1-c2)==1):   # c1 to the right
        dir1=West
        dir2=East
    elif ((r1-r2)==0) and ((c1-c2)==-1): # c1 to the left
        dir1=East
        dir2=West
    elif ((r1-r2)==1) and ((c1-c2)==0):  # r1 below
        dir1=North
        dir2=South
    elif ((r1-r2)==-1) & ((c1-c2)==0):    # r1 above
        dir1=South
        dir2=North
    else:
```

```
        raise ValueError,'Invalid neighboring indices'

    return dir1,dir2

def neighbors(idx,M,R,C):

    # convert the index to row and column
    r,c=idx2rc(idx,M,R,C)
    r=r[0]
    c=c[0]


    # get the neighboring row and column values

    rd=[v+r for v in [-1,0,0,1]]
    cd=[v+c for v in [0,1,-1,0]]

    # find the valid ones

    rdv=[]
    cdv=[]
    for i in range(4):
        if (rd[i]>=0) and (rd[i]<R) and (cd[i]>=0) and (cd[i]<C):
            rdv.append(rd[i])
            cdv.append(cd[i])

    # convert back to an index
    idx_neighbor=rc2idx(rdv,cdv,M,R,C)

    return idx_neighbor

def draw_row(row,sz):
    NEWS=num2news(row[0])

    #turtle starts in upper left, facing right
    if NEWS[2]:  #left-most wall
        right(90)
        pendown()
        forward(sz)
        backward(sz)
        left(90)
        penup()

    for v in row:
        NEWS=num2news(v)

        # I only care about the east wall and south walls
        # east
        penup()
        forward(sz)

        if NEWS[1]: # east wall
            right(90)
            pendown()
            forward(sz)
```

```
            backward(sz)
            left(90)
            penup()

        if NEWS[3]: # south wall
            right(90)
            forward(sz)
            right(90)
            pendown()
            forward(sz)
            backward(sz)
            left(90)
            penup()
            backward(sz)
            left(90)

def draw_row_text(row):
# draws one row of the maze

    NEWS=num2news(row[0])

    s=''
    if NEWS[2]:
        s=s+ '|' # left-most wall
    else:
        s=s+ ' '

    for v in row:
        NEWS=num2news(v)

        # I only care about the east wall

        if NEWS[1]:
            s=s+ '   |'
        else:
            s=s+ '    '
    print s

    # draw the south walls
    s='+'
    for v in row:
        NEWS=num2news(v)

        # I only care about the south wall

        if NEWS[3]:
            s=s+ '---+'
        else:
            s=s+ '   +'

    print s


def M2D(M1,R,C):

```

```
    # make 2D (all of the rc -> idx is down a column

    M=[]
    for r in range(R):
        M.append([0]*C)

    # copy
    count=0
    for c in range(C):
        for r in range(R):
            M[r][c]=Ml[count]
            count=count+1

    return M

def draw_maze_text(Ml,R,C):
    # Draws the maze in text form


    M=M2D(Ml,R,C)

    row=M[0]

    # draw the top wall

    s='+'
    for v in row:
        NEWS=num2news(v);
        if NEWS[0]:
            s=s+ '---+'
        else:
            s=s+ '   +'

    print s

    for row in M:
        draw_row_text(row)


def draw_maze(Ml,R,C):
    reset()
    speed(0)
    hideturtle()

    M=M2D(Ml,R,C)
    row=M[0]
    sz=10

    penup()
    x,y=-C*sz/2,R*sz/2
    goto(x,y)
    # draw the top wall
    for v in row:
        NEWS=num2news(v);
        if NEWS[0]:
```

```
                pendown()
                forward(sz)
            else:
                penup()
                forward(sz)

    penup()
    goto(x,y)
    for row in M:
        draw_row(row,sz)
        y-=sz
        penup()
        goto(x,y)

def maze(R=20,C=-1):
    #Aldous-Broder's algorithm

    #  Pick a point, and move to a neighboring cell at random. If an
    #  uncarved cell is entered, carve into it from the previous cell. Keep
    #  moving to neighboring cells until all cells have been carved into.

    if C<0:  # default value
        C=R


    M=[15]*(R*C)  # all walls up



    count=1

    # Pick a point...
    idx=random.choice(range(R*C))


    # Keep moving to neighboring cells until all cells have been carved into.
    while count<(R*C):

        #...and move to a neighboring cell at random.
        idx_neighbors=neighbors([idx],M,R,C)
        idx_new=random.choice(idx_neighbors)


        # If an uncarved cell is entered...
        if M[idx_new]==15: # uncarved
            # ...carve into it from the previous cell.
            dir1,dir2=get_direction([idx],[idx_new],M,R,C)
            M[idx]=M[idx]-dir1
            M[idx_new]=M[idx_new]-dir2

            count=count+1

        idx=idx_new


```

```
    return M


#==============================

R=60
C=60
M=maze(R,C)

M[R-1]-=1   # entrance
M[-R]-=8    # exit
draw_maze_text(M,R,C)


draw_maze(M,R,C)
```

Sample output is

```
>>> M=maze(10,15);
>>> draw_maze(M,10,15)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|     |  |        |        |     |        |  |
+  +  +  +  +--+--+--+--+  +--+  +--+  +--+  +
|  |     |                    |        |     |
+--+  +--+  +--+--+  +--+  +--+  +  +--+--+  +
|           |        |  |        |     |     |
+--+--+--+  +--+--+--+  +  +--+--+--+--+--+  +
|        |     |              |  |  |        |
+  +--+  +--+  +--+--+--+--+  +  +  +  +  +  +
|  |  |     |  |  |        |  |     |  |  |  |
+--+  +  +  +--+  +  +--+  +  +  +--+--+  +--+
|  |  |  |  |     |  |  |  |     |  |  |     |
+  +  +  +--+  +--+--+  +  +--+  +  +  +  +--+
|                                   |     |  |
+  +--+  +  +--+  +--+  +--+--+  +--+--+  +  +
|  |     |  |        |     |                 |
+--+--+--+  +  +--+--+--+--+--+--+  +--+--+--+
|           |                 |     |        |
+  +  +--+  +--+--+--+--+--+  +  +  +--+  +--+
|  |  |           |           |  |           |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Chapter 5

More Features of Python

5.1 Strings

Strings refer to strings of characters, like

```
In [1]: a='hello there'

In [2]: b="both types of quotes work"

In [3]: c=a+b

In [4]: print c
hello thereboth types of quotes work
```

5.2 File Input/Output

5.3 Tid-bits

In this section I am placing a number of useful techniques which don't fall easily into any other categories.

Swapping Two Values

If I want to swap the values of two variables, it is easiest done by the one-liner

```
>>> a=4; b=5;
>>> a,b=b,a
>>> a
5
>>> b
4
```

Looping through List Elements

One is often in the position of having to go through all of the elements of a list, to find the maximum or minimum, or perhaps a certain value. The straightforward way of doing it is a for-loop.

```
def find3(l):
    # find all of the elements that are equal to 3
```

```
    idx=[];  # start the index list equal to empty
    for i in range(len(l)):

        if l[i]==3:
            idx.append(i) # tack on the value i to the index vector

    return idx
```

used like

```
>>> l=[1,3,6,2,5,7,3,5]
>>> find3(l)
[1, 6]
```

Remember that indices start with 0!

Sorting

There are a number of ways of sorting a list of numbers. Some algorithms are very quick, but are more abstract to implement. A common algorithm which is not particularly fast, but is very easy to remember, is called the *bubble sort*. To sort in decreasing order, the bubble sort looks like:

- Go through each element, and compare it to the next element...
  - ... if the first element is smaller than the second, then swap them
- Repeat from the beginning, until you go through once and you haven't swapped any values

The code would look like

```
def sortit(x):
    # bubble sort
    y=x[:]  # make a copy of the list
    N=len(y)

    swapped=True
    while swapped:
        swapped=False

        for i in range(N-1):
            if y[i]>y[i+1]:
                y[i],y[i+1]=y[i+1],y[i]  # swap
                swapped=True

    return y
```

and run like

```
>>> l=[1,3,6,2,5,7,3,5]
>>> m=sortit(l)
>>> m
[1, 2, 3, 3, 5, 5, 6, 7]
```

Of course, there is already a function called `sort` which does just that.

```
>>> m=l[:]  # copy the list
>>> m.sort()
```

```
>>> m
[1, 2, 3, 3, 5, 5, 6, 7]
```