CHAPTER

1

Getting Started in Python

OUTLINE

1.1 Why Python? 4
1.1.1 *Comments* 5
1.1.2 *Errors* 6
1.1.3 *Indentation* 7
1.2 Numeric Variables 8
1.2.1 *Integers* 8
1.2.2 *Floating Point Numbers* 8
1.2.2.1 Built-in Mathematical Functions 9
1.2.3 *Complex Numbers* 10
1.3 Strings and Overloading 11
1.4 Input 14
1.5 Branching (If Statements) 14
1.6 Iteration 16
The Great Beyond 17
Further Reading 18
Problems 18
Short Exercises 18
Programming Projects 19
1. Harriot's Method for Solving Cubics 19

You can be shaped, or you can be broken. There is not much in between. Try to learn. Be coachable. Try to learn from everybody, especially those who fail. This is hard. ... How promising you are as a Student of the Game is a function of what you can pay attention to without running away.

***David Foster Wallace,* Infinite Jest**

CHAPTER POINTS

- Python is a computer programming language that we can use to solve engineering problems.
- One stores information in variables and can make computations and comparisons with those variables.
- Branching executes different parts of a code depending on conditions the programmer defines.
- Iteration execute the same block of code repeatedly under controlled conditions.

DOI: 10.1016/B978-0-12-812253-2.00002-9

1.1 WHY PYTHON?

In our study of computational nuclear engineering we are going to use Python, specifically version 3 of Python. Python is a powerful and widely accepted programming language that can do just about anything lower-level programming languages like Fortran, C, and C++ can. By learning to program in Python, you will learn the skills you need to program in any other computer language with relative ease.

While this book uses Python to explore computational nuclear engineering, it is not an exhaustive description of the Python language and how to use it. We will cover the topics needed for our computer simulation and numerical methods only. As a general computer programming language, Python can be used to analyze large data sets, write computer games, control devices, etc. The techniques we cover, and the approach we use to tackle problems using a computer will be applicable to these other fields as well.

The best way to start to learn a programming language is to actually use it to solve a problem. In almost any computer language the first thing you do is create a program called "Hello world!", where you make the computer say, in text, "Hello World!" (That is, after installing a way to write and run programs in the language. When installing Python on a machine, install Python 3 if you want to repeat the examples in this book. For those new to coding, a Python distribution such as Anaconda might be the easiest installation to begin with.) In Python you simply start a Python session and type:

```
In [1]: print("Hello World!")

Hello World!
```

This is the first command we will learn in Python, the `print` command:

BOX 1.1 PYTHON PRINCIPLE

The `print` command takes a comma-separated list of objects to print to the screen. Most commonly these are strings of characters contained inside either single quotes or double quotes. When printing to the screen, each object is separated by a space by default.

The code to type in to your Python interpreter is the part that follows `In [1]:` and the output is directly below it separate by a blank line. We could have it print any string of characters. The string of characters could be something simple such as

```
In [2]: print("Saw 'em off")

Saw 'em off
```

to more exotic characters:

```
In [3]: print("Søren Kierkegaard and Jean-Fraçois Lyotard")

Søren Kierkegaard and Jean-Fraçois Lyotard
```

Note how Python supports the unicode character set so that we can get those fancy characters. Actually typing those characters in from a standard US keyboard is trickier, but if you do manage to input them, Python can handle it (I used copy and paste).

These results were obtained by running the code in interactive mode via a Jupyter notebook, which means the result of each line is displayed when I enter the line, and the result of the last line of input is printed to the screen. It is more common to put your code into a separate file and then execute it. These files can be executed either on the command line by typing `python codename.py` where "`codename.py`" is the name of your file, or by running it in an integrated development environment, such as IDLE or Spyder.

1.1.1 Comments

A comment is an annotation in your code to

- inform the reader, often yourself, of what a particular piece of code is trying to do,
- indicate the designed output, result, etc. of a part of the code, and
- make the code more readable.

A comment can be anything to tell you or another reader of the source code what is going on in the code. Comments can also be useful to remind you to come back and clean up an ugly part of the code, or explain to your future self why the code is written in such a way.

Comments are your friend. They can be time consuming to add, but never have I looked back at old code and regretted adding them. Any code you write that other people *might* read should be well commented. This includes code you may write for a course on Python.

You use the pound (aka hashtag) # to comment out the rest of a line: everything that follows the # is ignored by Python. Therefore, you can put little notes to yourself or others about what is going on in the code.

```
In [4]: #This next line should compute 9*9 + 19 = 100
        print(9*9 + 19)

        #You can also make comments inside a line
        print(9*9 #+ 19)

100
81
```

There are also ways to make multiline comments by using a triple quote '''

```
In [5]: '''The beginning of a multiline comment.
        This comment will be followed by meaningless
        code.  Enjoy '''
        print("I am the very model of a modern major general.")

I am the very model of a modern major general.
```

BOX 1.2 PYTHON PRINCIPLE

A single line comment is everything on a line after a pound (or hashtag) character, #. Multiple lines can be commented by using triple quote at the start and at the end of the comment. Whatever is inside a commented block of code is ignored by the Python interpreter.

Later we will discuss some standard formats for comments at the beginning of a function. For now we will use comments as needed to illustrate what particular snippets of code are doing.

1.1.2 Errors

In any code you write longer than a few lines, you will make a mistake. In the parlance of our times these errors are called bugs. Now there are good bugs and bad bugs. (The term bug for an error or defect goes back to at least Thomas Edison in 1878 describing an error in an invention. The most celebrated use of the word was from Grace Hopper regarding an instance in 1947 when a moth lodged itself inside one of the components of the room-sized computers of the day, and caused a malfunction.) The good bugs get caught by Python and it will complain when it finds them. The bad bugs are insidious little beings that make your code do the wrong thing, without you knowing it. Good bugs are easier to find because Python will alert you to the error. Bad bugs can exist in a code for a long time (decades even) before being unearthed. Yes, decades: Microsoft Windows reportedly had a 17-year-old bug (http://www.computerworld.com/article/2523045/malware-vulnerabilities/microsoft-confirms-17-year-old-windows-bug.html).

Even experienced programmers write code with bugs. There are many different procedures to try to rid a code of bugs, but even the most sophisticated software quality assurance techniques will not catch every one.

We will now look at a good bug and a bad bug in the following code:

```
In [6]: #This is a good bug because the Python interpreter complains
        9*9 +


          File "<iPython-input-8-64b47963658c>", line 2
        9*9 +
            ^
    SyntaxError: invalid syntax
```

Notice that Python printed a whole host of mumbo jumbo to the screen, but if you look at it closely it tells you what exactly went wrong: in line 2 of the code, there was a plus sign without anything on the right of it. This bug is good because the code didn't run and you know to go back in and fix it.

A bad bug does the wrong thing, at least according to what you want it to do, and the user and the person writing the code may be none the wiser, as in this example:

```
In [7]: #This is a bad error because
        #it doesn't do what you might think
        #Say you want to compute (3 + 5)^2 = 8^2 = 64,
        #but you actually input
        print(3 + 5**2)
        #You don't get the correct answer,
        # and no one tells you that you're wrong.
```

```
28
```

This is an example of the power and feebleness of computers. A computer can do anything you tell it to, but it does not necessarily do what you want it to. Always keep this in mind: just because the computer did something, that does not mean it did what you wanted.

Later, we will talk in more detail about bugs, finding bugs (called debugging), and testing of code.

BOX 1.3 LESSON LEARNED

All codes that are longer than a few lines have bugs. This does not mean that those bugs meaningfully affect the program output, or that those bugs are ever encountered in the typical usage of the code. The bugs are there, however.

1.1.3 Indentation

Python is, by design, very picky about how you lay out your code. It requires that code blocks be properly indented. We will discuss what code blocks are later, but your code needs to be properly aligned to work.

```
In [8]: #If I improperly indent, my code won't work
        print("Not indented")
             print("indented, but it shouldn't be")
```

```
          File "<iPython-input-15-0b1b509e390c>", line 3
        print("indented, but it shouldn't be")
        ^
    IndentationError: unexpected indent
```

Notice that none of the code executed because of the indentation error. In Python when you indent something it tells the interpreter that the indented code is part of a code block that is executed differently than other levels of indentation. Only at certain times can one indent and it make sense. This sounds pretty abstract and nebulous right now, but it should become clear as we go through further examples.

1.2 NUMERIC VARIABLES

Almost every code needs to store information at some point in its execution. When this information is stored by a program in the computers memory, we call the identifier or name of the information a **variable**. Information, or data, is stored in variables using the equals sign. There are different types of variables for different types of data and we will discuss several of them here. Variable type means what type of information the variable stores. A simple example is storing a number versus text.

We will discuss numeric variables, i.e., variables that store a number, first. Later we will discuss how to store text and more exotic variables.

BOX 1.4 PYTHON PRINCIPLE

A Python expression of the form

```
variable_name = expression
```

will store in a variable named "variable_name" the value that expression evaluates to. The type of a variable indicates what kind of data the variable holds. The type function will identify a variable's type:

```
type(variable_name)
```

1.2.1 Integers

Integers are whole numbers, including the negatives. They never have a fractional, or decimal, part and should only be used for data that is a count.

```
In [9]: #assign the value 2 to x
        x = 2
        print(x*2)

        #check that x is an integer
        print(type(x))
```

```
4
int
```

The function `type(x)` returns the name of the type of the variable `x`. Notice that Python abbreviates the term integer to "int".

Integers are useful for things that can be counted: perhaps the number of times we execute a loop, the number of elements of a vector, or the number of students in a class.

1.2.2 Floating Point Numbers

Floating point numbers are numbers that do have a fractional part. Most numbers in engineering calculations are floating point types.

```
In [10]:#Now make some other floating point variables
        y = 4.2
```

```
        print("x =",x)
        print(type(y))
        #note that exponentiation is **
        z = (x / y)**3
        print("(2 / 4.2)**3 =",z)
```

```
x = 2
<class 'float'>
(2 / 4.2)**3 = 0.10797969981643449
```

The way that floating point numbers are represented on a computer has only a finite precision: there are only a finite number of bytes in the computer memory to hold the digits in the number. That means we cannot represent a number exactly in many cases. In fact floating point numbers are actually rational numbers (fractions) in the computer's internal workings. We will see later an example of how floating point accuracy can make a difference in a calculation.

1.2.2.1 Built-in Mathematical Functions

Having the ability to store floating point numbers and manipulate them with simple algebra would be of limited use to us without some common mathematical functions that we use repeatedly in engineering calculations. For instance, every time we wanted to evaluate the cosine function, we would have to program some approximation to the function, perhaps a Taylor series about a known value. Thankfully, almost every common mathematical function you might need is already built-in with Python. To use these functions you have to import the math functions using the command `import`.

BOX 1.5 PYTHON PRINCIPLE

In a code where you will be doing numerical calculations it is useful to start the code with `import math` to make a the wide range of common mathematical functions available by typing

```
math.[function]
```

where `[function]` is the name of the function.

In the code below, I set it up so that to use a math function you use the syntax `math.[function]` where `[function]` is the name of the function you want to call.

See https://docs.Python.org/3.4/library/math.html for the complete list of built-in mathematical functions.

The following code snipped uses the built-in Python function for computing the cosine of the number.

```
In [11]:import math
        #take cosine of a number close to pi
        theta = 3.14159
        trig_variable = math.cos(theta)
        print("cos(",theta,") =",trig_variable)
```

```
        #use the exponential to give e
        e = math.exp(1)
        print("The base of the natural logarithm is",e)

        #Python has a built-in pi as well
        print("The value of pi is",math.pi)

cos( 3.14159 ) = -0.9999999999964793
The base of the natural logarithm is 2.718281828459045
The value of pi is 3.141592653589793
```

Notice how in the print statements, if I give it multiple arguments, it prints each with a space in between. This is useful for combining static text with calculations, as we did above.

To evaluate logarithms we note an idiosyncrasy in the way that Python names the relevant functions. The natural logarithm is just `math.log` and the base 10 logarithm is `math.log10`.

```
In [12]:print("The natural log of 10 is",math.log(10))
        print("The log base-10 of 10 is",math.log10(10))

The natural log of 10 is 2.302585092994046
The log base-10 of 10 is 1.0
```

There are two non-obvious mathematical operators, integer division: `//`, and the modulus (or remainder): %

```
In [13]: # 7 / 3 is 2 remainder 1
         print("7 divided by 3 is",7//3,"remainder",7%3)
         print("851 divided by 13 is",851//13,"remainder",851%13)

7 divided by 3 is 2 remainder 1
851 divided by 13 is 65 remainder 6
```

1.2.3 Complex Numbers

Python can handle complex numbers, that is, numbers that have a real and imaginary part. We denote complex numbers using two floats: one for the real part and one for the complex part, multiplied by `1j`. The one is necessary so that Python knows you are not referring to a variable named `j`. Also, when Python prints complex variables, it typically surrounds them in parentheses. One can also do arithmetic with complex numbers using standard operators:

```
In [14]: z1 = 1.0 + 3.14 * 1j
         z2 = -6.28 + 2*1j
         print(z1,"+", z2,"=", z1+z2)
         print(z1,"-", z2,"=", z1-z2)
         print(z1,"*", z2,"=", z1*z2)
         print(z1,"/", z2,"=", z1/z2)

(1+3.14j) + (-6.28+2j) = (-5.28+5.140000000000001j)
(1+3.14j) - (-6.28+2j) = (7.28+1.1400000000000001j)
(1+3.14j) * (-6.28+2j) = (-12.56-17.7192j)
(1+3.14j) / (-6.28+2j) = (-0-0.5j)
```

To use common mathematical functions on complex numbers, we need to import the module `cmath`. With `cmath`, the common special functions and trigonometric functions can be applied to complex numbers. To illustrate this, we will compute the quadratic formula to find the roots of the polynomial

$$x^2 + \left(2 - \sqrt{2}\right) x - 2\sqrt{2} = \left(x - \sqrt{2}\right)(x + 2).$$

```
In [15]: import cmath
         a = 1.0
         b = (2 - math.sqrt(2))
         c = -2*math.sqrt(2)
         root1 = (-b + cmath.sqrt(b*b - 4*a*c))/(2*a)
         root2 = (-b - cmath.sqrt(b*b - 4*a*c))/(2*a)
         print("Roots are",root1,root2)

Roots are (1.4142135623730954+0j) (-2+0j)
```

Notice that this example used `cmath.sqrt` when taking the square root of a number that could be negative.

In `cmath` the constants `cmath.e` and `cmath.pi` are defined. We can use this to demonstrate Euler's famous relation:

```
In [16]: print(cmath.exp(cmath.pi*1j))

(-1+1.2246467991473532e-16j)
```

Here we see the effects of finite precision arithmetic in that this does not evaluate to exactly -1.

1.3 STRINGS AND OVERLOADING

A string is a data type that is a collection of characters, and needs to be inside quotes (you can use single or double quotes to enclose strings as the examples here will indicate):

```
In [17]: #This is a string
         aString = "Coffee is for closers."
         print(aString)
         print("aString")

Coffee is for closers.
aString
```

Anything inside the quotes is taken literally by Python. That is why the second print statement above just printed the literal text `aString`.

You can also subset, that is get some of the characters in a string, using brackets. Putting a single number in a bracket gives you the character in that position. Note, Python starts numbering at 0 so that 0 is the first character in a string.

BOX 1.6 PYTHON PRINCIPLE

A string is a collection of characters. The characters can be accessed individually using the string name followed by the character you want to access in square brackets. To access multiple characters use : indexing.

```
In [18]: aString[0]

Out[18]: 'C'

In [19]: aString[5]

Out[19]: 'e'
```

You can also get a range of characters in a string using the colon. The colon operator is non-intuitive in that $[a : b]$ says give me the elements in the string from position a to position $b - 1$.

- Python defines its ranges this way so that if the string is of length N, $[0 : N]$ returns the whole string.
- Negative subsets start at the end of the string (i.e., -1 is the last character of the string).

BOX 1.7 PYTHON PRINCIPLE

For strings the first character has index 0. To access multiple characters use the syntax

```
str_variable[a:b]
```

where `str_variable` is the name of the string and `a` is the index of first character returned and `b-1` is the index of last character returned. The first character in the string has an index of 0 and the last character has an index of -1.

Here are some examples of more advanced string indexing.

```
In [20]: aString[1:6]

Out[20]: 'offee'

In [21]:aString[-1]

Out[21]: '.'

In [22]: aString[-5:-2]

Out[22]: 'ser'
```

With characters (and strings) the + operator is overloaded. What we mean by overloaded is that the operator is defined so that the idea of what addition means is conferred to strings.

```
In [23]: 'Negative Ghostrider: ' + 'the pattern is full'

Out[23]: 'Negative Ghostrider: the pattern is full'

In [24]: 'a' + 'b' + 'c'

Out[24]: 'abc'
```

The + operator concatenates (or smushes together) the strings/characters it operates on. The multiplication operator is similarly overloaded, though it is not as general. It is simple to think about 'The roof,' * 3

```
In [25]: 'The roof, ' * 3

Out[25]: 'The roof, The roof, The roof, '
```

However, 'The roof,' * 3.15 is not defined and will give an error:

```
In [26]: 'The roof, ' * 3.15


---------------------------------------------------------------------
   TypeError            Traceback (most recent call last)

       <iPython-input-28-b68ac6dcc130> in <module>()
   ----> 1 'The roof, ' * 3.15


       TypeError: can't multiply sequence by non-int of type 'float'
```

The upshot is that only an integer times a character/string makes sense. The order of operations is respected by the operators

```
In [27]: 'The roof, ' * 3 + 'is on fire...'

Out[27]: 'The roof, The roof, The roof, is on fire...'
```

Minus makes sense, but only sometimes, so it is not allowed. In this instance, even though subtraction makes sense to us, Python does not allow it:

```
In [28]: 'The roof, ' - 'oof'


---------------------------------------------------------------------
   TypeError            Traceback (most recent call last)

       <iPython-input-30-c999e0505465> in <module>()
   ----> 1 'The roof, ' - 'oof'


       TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

The principle of operator overloading will be useful later when we talk about matrices and vectors. These objects have, for example, addition defined so that the user can add two vectors by using the plus sign +.

1.4 INPUT

There are times when you want the user of the program to interact with the program while it is running. For the purposes of engineering calculations our interactions will be fairly simple and through text input. The means that we can ask the user for input from the keyboard and record it. In Python we can prompt the user for input using the `input` command. To illustrate how this command works, we ask the user for a number and then double that number.

```
In [29]: user_value = input("Enter a number: ")
         user_value = float(user_value)
         print("2 *", user_value,"=", 2*user_value)

Enter a number: 9.95
2 * 9.95 = 19.9
```

When the program encounters an `input` command, it waits for the user to type something in and press the enter or return key. In this example, the user entered `9.95` as the input.

BOX 1.8 PYTHON PRINCIPLE

The function

```
input_variable = input(str_variable)
```

prints `str_variable` to the screen and waits for the user to enter a value. It will store a string of the characters the user entered in the variable `input_variable`. In Python 3 the input function always returns a string, even when the user enters a number.

Finally, this example also introduces the function `float`. This function takes a variable and changes it into a `float`. This is necessary because the `input` function always returns a character string variable.

1.5 BRANCHING (IF STATEMENTS)

Sometimes you want to execute code differently based on the value of a certain variable. This is most commonly done in if-else constructs. Here is an example that takes input from the keyboard and then executes different lines of code based on the response.

```
In [30]: instructors_op=input("What is your opinion of student? ")
         grade = ''
         if (instructors_op == 'annoying'):
             grade = 'F+'
         elif (instructors_op == 'Not annoying'):
             grade = 'B+'
         else:
             grade = 'A'
         print(grade)
```

```
What is your opinion of student? Not Annoying
A
```

What this codes says is that if the value of `instructors_opinion` is "annoying", the grade will be "$F+$", otherwise or else if (`elif` in Python-speak) `instructors_opinion` is "Not annoying" the grade will be "$B+$", and anything else will be a grade of "A". In the example I typed in "Not Annoying" and the `if` statement and the `elif` statement require that the string exactly match, so it executed the `else` part of the code.

BOX 1.9 PYTHON PRINCIPLE

The if-else construct allows the code to execute different branches based on the value of expressions. The code

```
if expression1:
        [some code]
elif expression2:
        [some other code]
else:
        [something else]
```

will execute the block of code `[some code]` if `expression1` evaluates to true, execute `[some other code]` if `expression1` evaluates to false *and* `expression2` evaluates to true, or will execute `[something else]` if both `expression1` *and* `expression2` evaluate to false. There could be more than one `elif` condition, or the `else` and `elif` statements could not be there at all. That is, it is possible to have an `if` without an `elif` or an `else`.

It is important to remember that when you want to check equality between two things you need to use == and not a single equals sign. A single equals sign is what you use when you want to assign something to a variable. You can compare numbers using the standard greater than, less than, and other operators. See Box 1.10 for a list of commonly used operators.

BOX 1.10 PYTHON PRINCIPLE

To compare numbers, and other variables, we can use the following operators to make comparisons:

- `a > b` — `a` greater than `b`
- `a >= b` — `a` greater than or equal to `b`
- `a == b` — `a` equal to `b`
- `a < b` — `a` less than `b`
- `a <= b` — `a` less than or equal to `b`
- `not(a)` — `a` not true

Each statement will evaluate to true or false.

In Python, when an expression evaluates to true, it evaluates to the integer `1`; a false expression evaluates to `0`. Therefore, we can treat a false expression as a zero and a true expression as non-zero, as we will do in later examples.

Python also has a `not` operator. This operator will return true if its argument is false (or zero); it will return false is the argument is true or nonzero. For example, `not(0)` will evaluate to true, and `not(1)` and `not(2.005)` will both evaluate to false. The not operator can be

combined with other expressions to make complex conditional statements. As an example, the mathematical statement $a \neq b$ can be written in Python as `not(a == b)`.

It is often common to have a condition where one checks if a number is close to another within some tolerance.

```
In [31]: import math
         pi_approx = 22/7
         if math.fabs(pi_approx - math.pi) < 1.0e-6:
             print("Yes, that is a good approximation")
         else:
             print("No,",pi_approx,
                   "is not a good approximation of",
                   math.pi,".")

No, 3.142857142857143 is not a good approximation of
3.141592653589793.
```

The function `math.fabs` is the float version of the absolute value function. In this case we were checking to see if an approximation is within 10^{-6} of π. Here the number 10^{-6} is written as `1.0e-6` which is shorthand for 1.0×10^{-6}.

Branching statements are most powerful when combined with iteration, as we will now explore.

1.6 ITERATION

Iteration executes a piece of code repeatedly, based on some criteria. In this example we will try to find a good approximation to π by trying many different values in succession.

```
In [32]: #this code finds a decent approximation to pi
         converged = 0
         guess = 3.14
         iteration = 0
         #Define tolerance for approximating pi
         eps = 1.0e-6
         #converged will be 0 if false, 1 if true
         converged = math.fabs(guess - math.pi) < eps
         while (converged == 0):
             guess = guess + eps/2
             converged = math.fabs(guess - math.pi) < eps
             iteration += 1 #same as iteration = iteration + 1
         print("Our approximation of pi is", guess)
         print("It took us", iteration,"guesses to approximate pi")

Our approximation of pi is 3.1415920000002227
It took us 3184 guesses to approximate pi
```

In this code, as long as `converged == 0` the code in the while block—the indented code below `while (converged == 0):`—will execute over and over. When the value of our

guess is within 10^{-6} to π in absolute value, `converged` will become 1 and the `while` loop will not start executing the code block again.

I did something tricky, but useful, in this example. In Python when a conditional expression like `a > b` is true it evaluates to an integer of 1, and evaluates to an integer of 0 when false. We will make use of this trick later on and it is good to see it now to help you get accustomed to it.

The idea of a while loop is not unique to Python, and can even be found in children's movies. The seven dwarfs in *Snow White* used the logic of a `while` loop to provide a soundtrack to their labors in a small mining operation, though they did not use Python:

```
while (working):
       [whistle]
```

BOX 1.11 PYTHON PRINCIPLE

The while loop is written in Python as

```
while expression:
        [some code]
```

This will execute the code in the block `[some code]` as long as `expression` evaluates to true when the loop returns to the top of the code block.

We can modify our code by tightening the tolerance to 10^{-8}, and we will change the condition for the `while` loop to show that there are multiple ways of accomplishing the same task.

```
In [33]: guess = 3.14
         iteration = 0
         eps = 1.0e-8
         converged = abs(guess - math.pi) >= eps
         while (converged==1):
             guess = guess + eps/2
             converged = abs(guess - math.pi) >= eps
             iteration += 1
         print("Our approximation of pi is", guess)
         print("It took us", iteration,"guesses to approximate pi")

Our approximation of pi is 3.141592644990321
It took us 318529 guesses to approximate pi
```

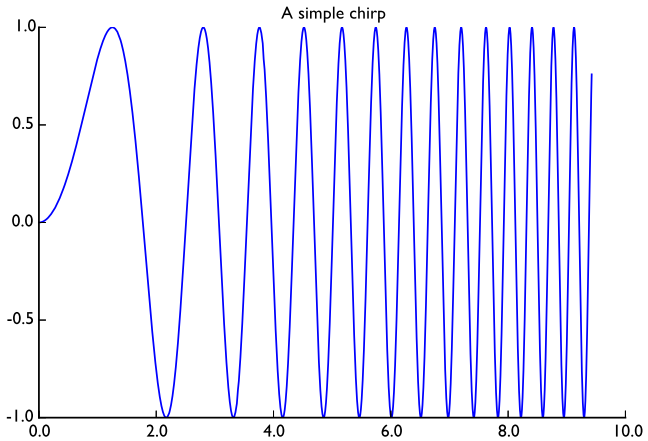
The `while` loop is an important form of iteration. Another type is the `for` loop which executes a set number of times based on a set condition. We will talk about that loop in the next chapter.

THE GREAT BEYOND

We have only scratched the surface of what Python can do. For example, we can generate graphs with relative ease, and we will cover this in detail in a few chapters. This will allow us

to visualize our calculations easily as both a check of our computation and a means to report our results. Here is an example of how simply we can generate the graph of a function:

```
In [34]: import matplotlib.pyplot as plt
         import numpy as np
         x = np.linspace(0, 3*np.pi, 5000)
         fig = plt.figure(figsize=(8,6), dpi=600)
         plt.plot(x, np.sin(x**2))
         plt.title('A simple chirp');
```



This example uses several features of Python that we have not discussed yet, namely `numpy` and `matplotlib`. These extensions of Python, called libraries, allow us to make a plot in just a few lines of code.

FURTHER READING

There are a number of reference books on programming in Python. An advanced text on using Python to write software to solve science problems is the work of Scopatz and Huff [1]. A more gentle introduction to Python and thinking like a coder is the work of Guttag [2].

PROBLEMS

Short Exercises

1.1. Ask the user for two numbers. Print to the user the value of the first number divided by the second. Make your code such that it never divides by zero.

1.2. Ask the user for an integer. Tell the user if the number is prime. You will need to use a while loop with a counting variable that goes up by one each time through the loop.

1.3. Ask the user for the length of two sides of a right triangle that form the right angle. Tell the user what the length of the hypotenuse is and the number of degrees in each of the other two angles.

1.4. When a object is not at rest, its mass is increased based on the formula $m = \gamma m_0$ where m is the mass, m_0 is the rest mass, and γ is a relativistic factor given by

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}.$$

The rest mass of a baseball is 145 g and the speed of light, c, is 2.99792458×10^8 m/s. What is the mass of a baseball when thrown at $v = 169.1$ km/h (the fastest recorded pitch)? How fast does it have to move to have a mass of 1.45 kg?

Programming Projects

1. Harriot's Method for Solving Cubics

The cubic equation

$$x^3 + 3b^2x = 2c^3$$

has a root given by

$$x = d - \frac{b^2}{d},$$

where

$$d^3 = c^3 + \sqrt{b^6 + c^6}.$$

This method was developed by Thomas Harriot (1560–1621), who also introduced the less than and greater than symbols.

Write a program that prompts the user for coefficients of a general cubic,

$$Ax^3 + Bx^2 + Cx + D,$$

and determines if the cubic can be solved via Harriot's method (i.e., $B = 0$). If it can be solved via Harriot's method, then print the solution. Also, print the residual from the root, that is the value you get when you plug each into the original equation. Make sure that you do not divide by zero, and that your method can handle imaginary roots.