Table of Contents

Chapter 3~~t3 [ChapterStart]~~

Arithmetic – Transforming Numbers

Arithmetic is what you think of when you think of math: adding, subtracting, multiplying, and dividing. With calculators and computers, arithmetic is pretty trivial.~~.~~ ~~S~~so in this chapter, we'll challenge ourselves to do even harder tasks. We'll use Python to

Basic Calculations

Doing arithmetic in the interactive Python shell is easy: you just ~~enter~~type the expression and press **ENTER** when you want to do the calculation.

Here's a table of the most common mathematical operators:

addition: +

subtraction: -

multiplication: *

division: /

exponent: **

Open your Python shell and try out the examples in Listing 3-1. :

```
>>> 23 + 56  # Addition
79
>>> 45 * 89  # Multiplication is with an asterisk
4005
>>> 46/13 # Division is with a forward slash
3.5384615384615383
>>> 2 ** 4    #2 to the 4th power
16
```

Listing 3-1: Basic math operators

The answer will appear as output. You can use spaces to make the code more readable (6 + 5) or not (6+5) and it won't make any difference to Python.

These are just examples of basic arithmetic, but Python has a lot more power than this. You can use variables to store numbers, and~~them: transform~~ to do operations on them no matter what their value is.

```
>>> x = 5
>>> x + 2
7
>>> length = 12
>>> x + length
19
```

In the above code, x starts with the value 5, then it's incremented by 2, becoming 7. The variable "length" is 12. When you add x and length, it's 7 + 12, which is 19.

shortcuts,Some operators have You'll come across There's a shortcut for incrementing a number.-. At times you'll The shortcut If you want to make a variable go up by 1, for example, no matter what the current value is, you can type . Instead of typing

```
x = x + 1
```

Yyou'll see Or you can use the shortcut

```
x += 1
```

Here the `x` is storing a value, then you're telling it to make that value one larger.

The same can be done And so on for the other operations, too

```
>>> y = 1
>>> y += 3
>>> y
4
>>> y -= 1
>>> y
3
>>> y *= 4
>>> y
12
>>> y /= 6
>>> y
2.0
```

Python can be a much more powerful calculator than anything you'll buy out there, and it's free! We're just getting started. There are a lot of repetitive tasks in arithmetic we can automate with Python, like finding averages, square roots, and factoring numbers.

Let Python do you Homework For You

There are a lot of repetitive tasks in arithmetic, and Python is excellent at automating repetitive tasks. We'll look at shortcuts for finding averages, square roots, and factoring numbers.

A Function to Find the Mean Average

One of the most common arithmetic tasks is finding the mean ~~average~~ of two numbers, done by adding the figures together and dividing them by the number of figures:

(10 + 20) / 2 = 15

.

The Average of Two Numbers Challenge~~Exercise 3.1~~

We'll write a Python program to ~~do~~ find the average of two numbers. You should be able to run the function ~~on~~and give it two numbers, and have it print the average, like this:

```
>>> average(10,20)
15.0
```

~~Solution:~~

The **~~"~~average~~"~~** function will transform two numbers, a and b, into half their sum. Then we'll return that value using the keyword "return." You might think this will do the trick:

```
def average(a,b):
    return a + b / 2
```

But when we test it, we get the wrong output:

```
>>> average(10,20)
20.0
```

That's because of the **order of operations**. It's dividing b by 2, then adding a. We need to use parentheses to add the two numbers first:

```
def average(a,b):
    return (a + b) / 2
```

Here's what happens when we run it:

```
>>> average(10,20)
15.0
```

Transforming Whole Numbers: The Running Sum

When you loop through a bunch of numbers, it's common to need to keep track of the running total of those numbers. The way to do this is to create a variable, which you might name `running_sum`, -

(remember, “sum” is taken already), set it to begin with a value of zero, and add to it each time a value is added. For this we use the `+=` notation again, for example:

```
>>> running_sum = 0
>>> running_sum += 3
>>> running_sum
3
>>> running_sum += 5
>>> running_sum
8
```

Remember that using the += command is a shortcut, so using `running sum += 3` is the same as `running_sum = running_sum + 3`. Let's increment the running sum by 3 a bunch of times to test it out:

```
running_sum = 0
for i in range(10):
     running_sum += 3
print(running_sum)
```

From reading this, you might be able to work out what the final sum is:

```
>>> 30
```

We're telling Python to first create a `running_sum` variable with the value `0`, then we tell it to run the for loop ten times, with the `(10)` argument to `range`. The content of the loop, indicated with the indentation, says to add three to the value of `running_sum` on each run of the loop. Once the loop has run ten times, Python will carry on with the code that comes next, in this case the `print` statement that displays the value of `running_sum` at the end of ten loops. Ten multiplied by three is thirty, so the output makes sense!

If you wanted to add all the numbers from 1 to 100, the range would be :

```
range(1,101)
```

Exercise 3.2:

Write a function called **“mySum”** that will take an integer as a parameter and return the sum of all the numbers from 1 up to that number, like this:

```
>>> mySum(10)
```

```
55
```

Because 1+2+3+4+5+6+7+8+9+10 = 55

Solution:

First you declare the value of the running sum and then you increment it in the loop:

```
def mySum(num):
    running_sum = 0
    for i in range(1,num+1):
        running_sum += i
    return running_sum
```

Using a variable we can save one number, but what if we want to save more than one number? We'll use a list.

Lists

A lot of math involves repetition, like factoring a number, and a good way to save time and energy is to create variables that can hold more than one value, known as *lists*. We've learned how to save one value to a variable, but how can we save more than one value? To declare a list in Python you simply create a name for the list, use the `=` command like you do with variables, and then enclose the items you want to place in the list in square brackets, each separated by a comma, like this:

```
>>> a = [1,2,3]
>>> a
[1, 2, 3]
```

Often it's useful to ~~is~~ create an empty list so you can add values to it afterwards. Here's how:

```
>>> b = []
>>> b
[]
```

This list exists, but has nothing inside. Let's see how to add things.

Appending to a List

To add an item to a list you use the `append` command. First name the list you want to add to followed by a period, then use `append` with the item you want to add as an argument in parentheses.

```
>>> b.append(4)
```

```
>>> b
[4]
```

You can see the list now contains just the figure `4`. You can add items to lists that aren't empty, too.

```
>>> b.append(5)
>>> b
[4, 5]
>>> b.append(True)
>>> b
[4, 5, True]
```

The item appears at the end of the list. You can also add text as strings. This time the text you want to include needs to have either double or single quotes around it.

```
>>> b.append("hello")
>>> b
[4, 5, True, 'hello']
```

Removing an item is just as easy: instead of `append`, use `remove`, with the item you want to remove as the argument. Make sure to get the item you're removing exactly the same as it is in the code, or Python won't understand what to delete.

```
>>> b.remove(5)
>>> b
[4, True, 'hello']
```

This removed `5` from the list, and notice that the rest of the items stay in the same order. That will become important later.

Lists in Loops

If you want to perform the same action on each item in a list, you can iterate over all the elements using a `for` loop. The iterator, the `i` variable in `for i in range(10)` can be called anything you want, for example:

```
>>> a = [12,"apple",True,0.25]
>>> for thing in a:
        print(thing)
```

```
12
apple
True
0.25
```

Here we just print each item to the screen. Notice that they're printed in order, and that each item is on a new line. If you want to print everything on the same line, you need to add an “end” to your print function:

```
>>> for i in range(5):
        print(i,end='')
01234
```

Using List Indexes

You can refer to any element in a list by specifying the name of the list and then using its **index**~~, also~~ in square brackets:

```
>>> b
[4, True, 'hello']
>>> b[0]
4
>>> b[2]
'hello'
```

Remember the indices start at 0, not 1, so the index of the first element in `0`.

The `:` syntax lets you access a range of elements. For example, if you want everything from the second item of a list to the 5th, for example, the syntax would be:

```
>>> myList = [1,2,3,4,5,6,7]
>>> myList[1:6]
[2, 3, 4, 5, 6]
```

If you don't specify the ending index of the range, Python will return all elements from the first index onward. For example, ~~Y~~you can access everything from the second element (index 1) onward with:

```
>>> b = [4, True, 'hello']
>>> b[1:]
```

```
[True, 'hello']
```

A very useful thing to know is that you can access the last terms in a list even if you don't know how long it is by using negative numbers. To access the last item, you'd use `-1`, for example:

```
>>> b[-1]
'hello'
>>> b[-2]
True
```

Back to the Average

Let's build a function that uses lists to find the average of a bunch of numbers. All we need is the sum of the numbers and how many numbers we're taking the average off.

First, we need to find out how many numbers are in a list, so let's create that as a separate function.

Counting the Items in a List

We'll use the running sum to count the number of items in a list. We'll call the function `countItems`, and it should take the name of the list it's averaging as the argument. When we use the function, it should look like this:

```
>>> a = [12,"apple",True,0.25]
>>> countItems(a)
4
```

First we'll define the function. It will take a list as a parameter. The first thing we'll do is declare a variable for the count.

```
def countItems(listName):
    count = 0
```

Next we'll iterate over the items in the list and add 1 to the running sum for each item:

```
    for item in listName:
      count += 1
```

Finally we'll return the count:

```
    return count
```

Now it's up to you to use what you know and have so far the build the rest of the function!

Exercise 3.4: Finding the Average

Use **`countItems`** to create an **`averageOfList`** function that will find the average of a list of numbers.

Solution:

Here's how, using running_sum and countItems:

```
def averageOfList(numList):
    running_sum = 0
    items = 0
    for item in numList:
        running_sum += item
    return running_sum / countItems(numList)
```

Here's ~~an example of~~ how you would use the function:

```
>>> averageOfList([8,11,15])
11.333333333333334
```

Python Counting Keywords

Building a function from scratch is a great exercise, but there are actually built-in keywords in Python for summing the numbers in a list, `sum`, and finding the length of a list, `len`, that can make this much easier.

```
>>> sum([8,11,15])
34

>>> len([8,11,15])
3
```

You simply enter the keyword and pass the list as the argument. A more concise version of the average function would be :

```
def average3(numList):
    return sum(numList)/len(numList)
```

Making Comparisons with Conditionals

~~Here's a~~You can have your programs make their own decisions using another vitally important programming tool: ***conditionals***. A conditional makes comparisons in your programs, allowing you to use the result to decide what to do next.

~~, or "if" statements. They check whether a statement is true, and if it is, the program executes some code. If it's false, the program might do something else or do nothing. `True` and False are capitalized in Python~~ True and False are values in Python... For example:

```
>>> y = 6
>>> y > 5
True
>>> y > 7
False
>>> y == 6
True
```

We set the variable `y` to contain 6, and then make three statements about the variable `y`. The first claims that the value of `y` is bigger than 5, which is true, so Python returns `True`. The second claims y is bigger than 7, which is `False`, and the third says `y` is equal to 6, also true. This is how Python makes comparisons.

Making Decisions with if

You can make your program make decisions about what code to run by using conditionals with `if` statements. If the condition you set turns out to be true, the program runs one set of code. If the condition turns out to be false, the program might do something else or do nothing at all.

Here's an example:

```
>>> if y > 5:
      print("yes!")

yes!
```

Here we are saying: if the value of `y` is more than 5, print “yes!”; otherwise do nothing.

Rather than having it do nothing, Yyou can give other your programs alternative code to run s, too, using when a value is false using "**else**":

```
y = 6
if y > 7:
    print("yes!")
else:
    print("no!")
```

Here we say: if the value of y is more than seven print `yes!`, otherwise print `no!`. Run this, and it will print "no!".

You can add more Other alternatives usinge "**elif**," iswhich short for "else if." You can have as many **elif**'s as you want. Here's an example program:

```
age = 50
if age < 10:
    print("What school do you go to?")
elif 11 < age < 20:
    print("You're cool!")
elif 21 < age < 30:
    print("What job do you have?")
elif 31 < age < 40:
    print("Are you married?")
else:
    print("Wow, you're old!")
```

This program runs different code depending on which of the specified ranges the value of `age` falls in. When `age = 50`, Tthe output will be:

```
Wow, you're old!
```

I'm sure you can imagine how this could be endlessly helpful!

Checking a List with in

One handy use ofA really useful thing you can do with **conditionals** is in checking whether a value is in a list contains a particular value using "**in**. When you have a really long list, it can be a pain to check through each value manually for a particular item. With `in` you give the value you're looking for, then the keyword `in`, and then the list you are checking, and Python will tell whether the statement is

~~"boolean," an expression that will return Like every conditional statement, this returns a "~~True or False with a boolean value.

```
>>> b = [1,2,3,'hello']
>>> 1 in b
True
>>> 4 in b
False
>>> "hello" in b
True
```

~~And i~~If a value is in the list, you can find its index by asking for it with the list name followed by the `.index` syntax, giving the value you are searching for as an argument:

```
>>> b.index(1)
0
>>> b.index('hello')
3
>>> b.index(4)
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    b.index(4)
ValueError: 4 is not in list
```

~~Read~~ The last attempt gives us an error message. The last line in an error message is usually the one that tells you the cause of the error: ~~s from the last line first.~~ This one says ~~on the last line~~ that the value we are looking for, **`4`**, is not in the list, and so Python can't give us its index.

Factors of a ~~n~~Number

~~Finally~~Now~~,~~ ~~we~~you're going to use what you've learned so far to factor a number! A ~~number is a~~ factor is a number that ~~of another if it~~ divides evenly into ~~it~~another number; for example, 5 is a factor of ten because 10 can be divided evenly by 5. In Python ~~(and many other languages) use the~~ you use the modulo operator, ~~the percent sign (`%`)~~, to calculate the remainder when two numbers are divided. So if **`a % b`** ~~is~~equals zero, it means ~~then `b`~~ divides evenly into **`a`**. Here's an example of the mudolo in action:

```
>>> 20 % 3
2
```

This means shows that when you divide 20 by 3, you get a remainder of 2, meaning that 3 is not a factor of 20.. If we try five, however:

```
>>> 20 % 5
0
```

We get a remainder of 0, sSo 5 is a factor of 20.>>> 21 % 5
1

5 is not a factor of 21.

Let's put thatthis to use together into a function that will take a number and return a list of that number'sits factors. Why don't we we could instead just print out the factors, but ? Well, we're going to use the factors list later in another function, to find the greatest common factor.

When starting a new program, it can be good to lay out your planThe plan is:

1. Define a `factors` function that takes a number as an argument
2. Create an empty factors list to fill with factors
3. Loop over all the numbers from 1 to the argumentat number
4. If any divide evenly, add it to the factors list
5. Return the list of factors at the end.

Here's the function. Enter this into a new file in IDLE and save it as *factors.py*:

```
def factors(num):
    '''returns a list of the factors of num'''
    facts = [] #empty factor list
    for i in range(1,num+1): #go from 1 to num
        if num % i == 0: #if i divides evenly
            facts.append(i) #it's a factor, so add it to the list
    return facts
```

Now run *factors.py* and you can use the `factors` function in the normal IDLE terminal by passing it a number to find the factors for. Here are all the factors of 120:

```
>>> factors(120)
[1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
```

We can use that this function to find the **greatest common factor** of two numbers.

ExerciseThe Greatest Common Factor Challenge 3.5:

Our challenge is to ~~W~~write a program to find the greatest common factor of two numbers.

***Exercise 3.5**: Write out a plan for this program like we did for the `factors` function. What new steps will you need? What new challenges does this program present?*

~~Solution:~~

We'll need our this program to be able to calculate the factors of each number, so open *factors.py* and we'll build from there. To find the ~~"~~greatest common factor (GCF)~~"~~ of two numbers, our function needs to be able to accept two numbers.

```
def greatest_common_factor(a,b):
```

We create the `greatest_common_factor` function that will take two separate numbers as its argument.

We then need the factors of each number `a` and `b` so we can compare them and find the factors common to both lists. We'll create the lists `afacts` and `bfacts` to store the factors.

```
    afacts = factors(a)
    bfacts = factors(b)
```

Inside these lists we are applying the `factors` function we made earlier to the variables `a` and `b`, so that the lists will contain the factors of `a` and `b`. Now we need a way to compare the two lists and find the biggest factor that's common to them both. To find the biggest

~~of 24 and 60, for example, we would factor the numerator and denominator, then choose the largest number which is a factor of both would be our GCF.~~ One way to do this is to reverse the factor list so you're starting from the largest one and then compare them, then print the first factor we find that's common to both lists. That way you can trust that the first common factor is also the largest common factor. ~~This is done~~ We can reverse the list with the ~~"~~**~~reverse~~**~~"~~ function ~~in the code below~~built into Python.

```
~~def greatest_common_factor(a,b):~~
~~    '''finds the greatest common factor of two numbers'''~~
~~    #get lists of factors of each number~~
~~    afacts = factors(a)~~
~~    bfacts = factors(b)~~
```

```
    #reverse one list of factors:
    afacts.reverse()
```

Now when we cycle through the list, we'll be going in descending order.

```
  #go through list, greatest first
    for fact in afacts:
        #if one of the factors is in the other list
        if fact in bfacts:
            #it must be the greatest
            return fact
```

We use a `for` loop to to check each value (`fact`) in `afacts`. We then use our new skills with `if` and `in` to check, for each value, if that same value is also in `bfacts`. If it is, we print it to the console and the program ends, ensuring only the largest common factor is printed. Notice the indentation levels here: our second `if` statement is inside the for loop.

Here's how we can use the function to get the greatest common factor of 54 and 24:

```
>>> greatest_common_factor(54,24)
6
```

An otherwise laborious task done in a second with Python!

Transforming Whole Numbers II: Numbers (Getting) Big and Small

This next challenge ~~Here's~~ a good exploration into exponential growth: ~~how surprisingly quickly~~ numbers ~~can get big, and how they can get small~~growing and shrinking at a surprising rate.

~~The Penny-A-Day Challenge~~~~Exercise 3.6:~~

~~Imagine you have a job~~ Let's say you've been offered a job that entails ~~to do backbreaking work,~~ 8 hours of manual labor a day for 30 days. The employer offers you two salary options:~~.~~ ~~Y~~you can receive a flat $25 an hour rate for the whole month, or you can make 1 penny for the first day and 2 pennies the next day, and 4 pennies on the next, doubling your daily wages every day. Which should you choose? Let's compare the net cash received at the end of both options.

~~Solution:~~

Calculating The Rates

The first option is easy to calculate using straightforward multiplication:

```
>>> 25 * 8 * 30
6000
```

You'll make $6,000 for the month. That's ~~N~~not bad, but ~~what about the~~ we should check out the other option before making a decision.~~? A penny a day isn't even "working for pennies" but b~~B~~y day ~~2~~two ~~if you've saved all your earnings you have~~ you will have earned 3 cents total; by day three it'll be~~.~~ ~~7~~ Seven cents total. Seven cents after three days ~~3~~ of (need I remind you?) backbreaking work and you might think you've made a terrible mistake. But don't be fooled! ~~if w~~We can write a program using a ~~running sum, we can~~ to find out how much this will finally add up to:

```
def penny(n):
    '''Calculates the sum of working
    for n days.'''
    running_sum = 0
    wages = 0.01 #first day's wages
    for i in range(n):
        running_sum += wages
        wages *= 2 #double the daily wages
    print(running_sum)
```

To use the program, we pass the function `penny` the number of days to sum the running total of. ~~After~~ For example, after day 1 our running total will be:

```
>>> penny(1)
0.01
```

After day 3:

```
>>> penny(3)
0.07
```

Check. So let's skip up to day 30:

```
>>> penny(30)
10737418.23
```

That's over 10 million dollars! So ~~I~~you'd starve the first week, but after a while the numbers grow~~, well,~~ exponentially. When you're doubling, small numbers grow surprisingly quickly.

~~Halving a Great Time~~

~~Exercise Guessing the Numbers Challenge 3.7:~~

Now let's look at numbers shrinking exponentially with a number guessing challenge. I'm thinking of a number between one~~1~~ and a ~~million~~hundred, h~~H.~~ow many guesses do you think you'd~~would you~~ need to guess my number correctly? To narrow it down, after each incorrect guess ~~What if~~ I ~~had to~~ tell you whether your guess is ~~"~~higher~~"~~ or ~~"~~lower~~"~~ ~~each time you got it wrong?~~than the answer. Would it still take a long time? In face, there's a method you can use that involves halving that will make this task infinitely easier.

~~The opposite of doubling might be called "halving," and when you cut big things in half over and over, they get small surprisingly quickly. You may have figured out the strategy if I have to tell you to guess higher or lower:~~ When you make an incorrect guess, you're next guess should be ~~in~~ the middle ~~of the range~~number between your guess and the maximum (if your guess was too low) or minimum (if your guess is too high) it can be. That wipes out half the possible numbers every time, and you'll be surprised how quickly it hones in on the answer. ~~So, how many guesses? Even a ballpark estimate?~~ Let's take this one step at a time.

~~Solution~~Making a Random Number Generator

~~Let's take this one step at a time.~~ First we need the computer to choose a number at random between 1 and 100~~(and don't tell us what it is!)~~. We do that by importing the random module and assigning a random integer to a variable. Create a new file in IDLE and save it as *numberGame.py*, then enter the following:

```
from random import randint

def numberGame():
    #choose a random number
    #between 1 and 100
    number = randint(1,100)
```

This creates a number variable that will hold a random number between 1 and 100, generated each time we call it.

Taking User Input

Now the program ~~has to~~ needs to ask the user for input so the user can~~:~~ take a guess! ~~This is done with~~ We'll use the input function you learned about it Chapter XX~~. The program will print out what's in the parenthese (in Python 2 it's "raw input")~~s:

```
>>> name = input("What's your name? ")
```

```
What's your name?
```

The text that you add into the parentheses will be printed in the terminal when you call `name`, asking the user to input their name. The user ~~types in~~ enters something, presses Enter, and the program saves it.

```
What's your name? Peter
>>> print(name)
Peter
>>>
```

If we save this input into a variable, we can use it later in our program:~~You can do something with the input of course:~~

```
def greet():
    name = input("What's your name? ")
    print("Hello,",name)

greet()
```

The output will be:

```
>>>
What's your name? Al
Hello, Al
>>>
```

Input from users is always taken in as a string in Python, so if we're dealing with numbers we have to convert the input to an integer so we can use it in operations.

```
print("I'm thinking of a number between 1 and 100.")
guess = int(input("What's your guess? "))
```

We add `int` and make the input request an argument of `int`, and when the user enters input it will be transformed into an integer.

START BOX

Exercise 3.8:

Write a short program that will take the user's name as input, and if they enter “Peter” it will print “That's my name, too!” If the name is not “Peter” it will just print “Hello,” and the name.

Solution:

```
name = input("What's your name?")
if name == "Peter":
    print("That's my name, too!")
else:
    print("Hello, ", name)
```

~~By the way, n~~Notice that you use the double equals sign when checking for equality. Single equals signs means you're assigning a value to a variable.

End box

~~Back to the number guessing game.~~

Checking for a Correct Guess

~~We~~ Now the program needs a way to check if the number the user guessed is correct. If it is, we'll announce the guess is right and the game is over. Otherwise, we need to tell the user ~~if~~whether they should guess higher or lower. We'll use the `if` statement to compare the input the to content of `number`, and use `elif` and `else` to decide what to do in each circumstance. Add the following to your program.

```
if number == guess:
        print("That's correct! The number was",number)
    elif number > guess:
        print("Nope. Higher.")
    else:
        print("Nope. Lower.")
```

If the random number held in `number` is equal to the input stored in `guess`, we tell the user their guess was correct and give them the random number. Otherwise, we tell them whether they need to guess higher or lower.

Here's an example of the output so far:

```
I'm thinking of a number between 1 and 100.
What's your guess? 50
Nope. Higher.
```

~~Then it ends~~Pretty good, but currently our program ends here and so doesn't let the user make any more guesses.

Guess Again!

We need to make a loop so ~~it~~ the program keeps asking for more guesses until the user guesses correctly. ~~“~~We'll use the ~~W~~w**hile** ~~guess”~~ loop to~~will~~ keep looping ~~as long as the user enters a guess. If~~ until ~~the~~ **guess** is equal to ~~the~~ **number**~~.~~, and then ~~it~~ the program will print~~s~~ out a success message and ~~“~~break~~s”~~ out of the loop.

```
while guess:
        if number == guess:
            print("That's correct! The number was",number)
            break
        elif number > guess:
            print("Nope. Higher.")
        else:
            print("Nope. Lower.")
        guess = int(input("What's your guess? "))
```

The Halving Guessing Method

Save the program and you're ready to run it. ~~Now, by running the program, we'll~~ Use the halving method, where each time you make an incorrect guess your next guess is exactly half way between your first guess and the closest end of the range. For example, if you start by guessing 50 and the program tells you to guess higher, your next guess would be half way between 50 and 100 at the top of the range, so you'd guess 75. ~~s~~See how many guesses it'll take to guess a number between 1 and 100. Figure 3-1 shows an example.

```
from random import randint

def numberGame():
    #choose a random number
    #between 1 and 100
    number = randint(1,100)
    print("I'm thinking of a number between 1 and 100.")
    guess = int(input("What's your guess? "))
    while guess:
        if number == guess:
            print("That's correct! The number was",number)
            break
        elif number > guess:
            print("Nope. Higher.")
        else:
            print("Nope. Lower.")
        guess = int(input("What's your guess? "))

numberGame()
```

```
I'm thinking of a number between 1 and 100.
What's your guess? 50
Nope. Lower.
What's your guess? 25
Nope. Lower.
What's your guess? 12
Nope. Lower.
What's your guess? 6
Nope. Higher.
What's your guess? 9
Nope. Higher.
What's your guess? 10
That's correct! The number was 10
>>>
```

Figure 3.1: The code and the output of the number guessing game.

This time it took 6 guesses. $2^6 = 64$ and $2^7 = 128$ so it makes sense that on average it takes around 6 or 7 guesses.

Exercise 3.9:

Write a program that will guess the number you're thinking, ~~s~~ using the halving method ~~above rules~~. Use our `average` function to keep narrowing the range as guesses are made. As an example, I chose 4 as my mystery number, and this is how the program to figured it out:

```
Think of a number between 1 and 100.
Enter c if I'm right, h if I should guess higher
and the letter l if I should guess lower.
My guess is 50
l
My guess is 25
l
My guess is 12
l
My guess is 6
l
My guess is 3
h
My guess is 4
c
```

the program got it in six guesses, not bad! We'll use this halving method to find solutions of algebraic equations in the next chapter.

Solution:

~~We'll use our average function and keep narrowing the range as we make guesses:~~

```
def average(a,b):
    return (a + b)/2

def guessing(lower,upper):
    '''makes guesses between lower and upper'''
    print("Think of a number between {} and {}.".format(lower,upper))
    print("Enter c if I'm right, h if I should guess higher")
    print("and the letter l if I should guess lower.")
```

```
    while True:
        #the guess has to be an integer
        guess = int(average(lower,upper))
        print("My guess is", guess)
        response = input()
        if response == 'c':
            break
        elif response == 'h':
            lower = guess + 1
        else:
            upper = guess - 1

guessing(1,100)
```

> I chose 4 as my mystery number. Let's see how long it takes my program to figure it out. This is the output:

```
Think of a number between 1 and 100.
Enter c if I'm right, h if I should guess higher
and the letter l if I should guess lower.
My guess is 50
l
My guess is 25
l
My guess is 12
l
My guess is 6
l
My guess is 3
h
My guess is 4
c
```

> Six guesses, not bad! We'll use this "halving" method to find solutions of algebraic equations in the next chapter.

Transforming Rational and Irrational Numbers:

This section will give you a few functions you can make to do you fractions homework for you. We'll look at functions for reducing, multiplying, and adding fractions, and a function to find square roots. These functions use only the tools we've covered so far, so see if you can work out how each one accomplishes its goal.

Reducing Fractions

Reducing fractions into their smallest form is a common math class task, so let's create a function to do it for us. The ~~greatest common factor is a~~ key tool in reducing fractions is the greatest common factor. ~~All you need is~~ When you have the GCF, you can just ~~of the numerator and the denominator~~ and divide the numerator and denominator by that factor until it will divide no more, and that's your reduced fraction. Here's a function to do that:

```
def reduce(numerator, denominator):
    '''reduces the fraction numerator/denominator to its lowest form'''
    gcf = greatest_common_factor(numerator,denominator)
    numerator /= gcf #divide the numerator by the GCF
    denominator /= gcf #and the denominator
    return int(numerator), int(denominator)
```

To use the reduce function, enter the reduce keyword and give the numerator at the first argument and the denominator as the second argument. To reduce the fraction 24/54 you would enter:

```
>>> reduce(24,54)
(4, 9)
```

In its lowest form, $\frac{24}{54} = \frac{4}{9}$.

If the fraction is already in its lowest form, its ~~gcd~~ GCD is 1, so there's no change and the output will be the same as the input:

```
>>> reduce(5,4)
(5, 4)
```

Multiplying Fractions

Multiplying fractions is ~~easier than adding fractions~~simple. You just multiply the numerators, multiply the denominators, and reduce ~~if~~ the outcome if necessary. Here's a function ~~to do that~~for multiplying fractions:

```
def multiply_fractions(n1,d1,n2,d2):
    '''multiplies n1/d1 * n2/d2'''
    numerator = n1 * n2
    denominator = d1 * d2
    return reduce(numerator,denominator)
```

Now, to multiply $\frac{1}{2} \cdot \frac{3}{4}$, just enter

```
>>> multiply_fractions(1,2,3,4)
(3, 8)
```

Adding ~~f~~Fractions

Adding fractions is ~~harder~~more complicated than multiplying them. First you have to find a common denominator, which ~~is~~ will be the lowest common multiple (LCM) of the two original denominators. We need a Lowest Common Multiple function!

~~Exercise 3.10:~~The Lowest Common Multiple Challenge

We'll write a function that will take two numbers and return their lowest common multiple.

~~Solution:~~

To find the LCM of a and b, you should choose one of the numbers, let's say a, multiply it by 1,2,3 and so on until you get to a number that b divides evenly into. They definitely have a common multiple, a times b, so you won't have to go any higher than that.

```
def lowest_common_multiple(a,b):
    '''returns the LCM of two numbers'''
    mult = a #start with one number
    counter = 1 #we'll multiply by this number
    while mult <= a*b: #this is the largest possible LCM
        mult = a*counter #use all the multiples of a
        if mult % b == 0: #if it's also a multiple of b
            return mult
        counter += 1 #if not, go to the next multiple of a
```

Then you multiply the numerators by the LCM divided by their denominators. Then add the numerators. Finally, reduce the fraction if necessary. Here's the whole code:

```
def add_fractions(a,b,c,d):
```

```
    '''adds a/b + c/d'''
    #first find LCM of denominators
    lcm = lowest_common_multiple(b,d)
    #multiply numerators:
    a = int(a*lcm/b)
    c = int(c*lcm/d)
    sum_numerator = a + c
    return reduce(sum_numerator,lcm)
```

~~Transforming Irrational Numbers:~~ Square Roots

A good use of the average is approximating square roots. Say you don't know the square root of 60 but you guess it's 7.5. Average 60/7.5 and 7.5 to get your next guess:

```
>>> average(60/7.5,7.5)
7.75
```

Now plug the new guess in:

```
>>> average(60/7.75,7.75)
7.745967741935484
```

Let's automate this process:

```
def squareRoot(num, guess):
    '''approximates the square root
    of num by iteration. Guess is a
    first guess.'''
    for i in range(10):
        newguess = average(num/guess, guess)
        guess = newguess
    return guess, guess**2

>>> squareRoot(60,7.5)
(7.745966692414834, 60.00000000000001)
```

Even a terrible first guess can be corrected by all that iterating. After 6 iterations it's accurate to 5 decimal places. I changed the function to print out each guess.

```
>>> squareRoot(60,1)
30.5
```

```
16.233606557377048
9.964821455916496
7.993001548146788
7.749784170716252
7.745967632643636
```

Tools We Learned

The new tools we learned in this chapter were Conditionals (“if” statements), Lists, Input and booleans. They're going to be very important tools in our programs!