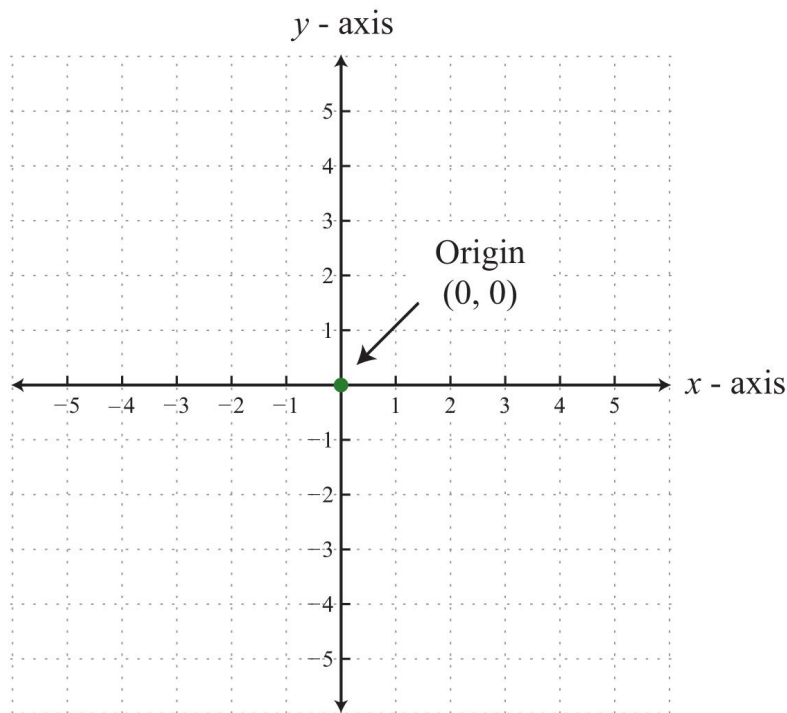


In this lesson, we are going to study chaos and order. While they intuitively mean the complete opposite of each other, we will find out that there is in fact a very close relationship between the two. Often, if you look closely enough at something apparently chaotic, you may find that there is some order to it after all.

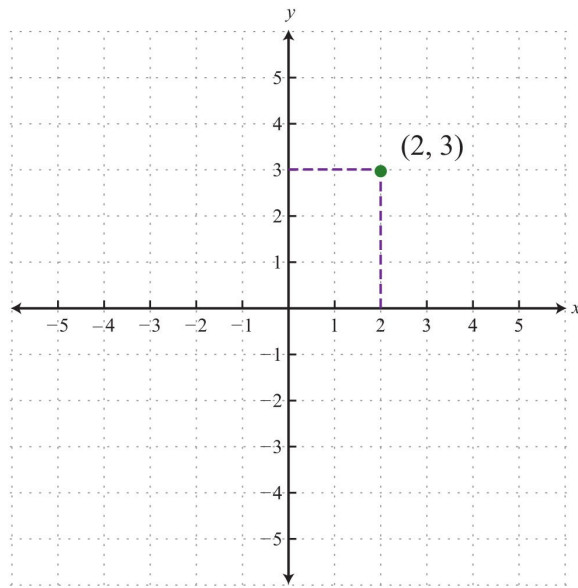
The coordinate system

A **coordinate system** allows us to represent positions of a specified dimension in a reproducible fashion. This means that if someone describes a position to us using a coordinate system, we can accurately represent that position. For example, the two-dimensional coordinate system allows us to represent points on a flat plane (similar to the page of a book or a white board on a wall). The system is made up of two axes that are perpendicular (i.e., at 90 degrees) to each other, and which meet at a point referred to as the **origin**. The position of any point on that plane is described using two values that describe how far away from the origin the given point is along both predefined axes. We typically refer to these axes as X and Y , with the X -axis oriented horizontally (i.e., left-to-right), and the Y -axis oriented vertically (i.e., up-to-down). Although this represents a way to locate points in two dimensions, we can extend this to more dimensions. For example, we can add an axis oriented at an angle (technically, you can think of it as being perpendicular to both the X - and Y -axes and going through the page or the white board). This adds a third dimension to the coordinate system and allows the location of objects in three-dimensional space.

Each axis in a coordinate system has values along it that demarcate one-dimensional positions. Although we have freedom in what those values are, we typically center the origin at the X -value 0 and the Y -value 0. We call this a point. The origin, then, rests at point $(0, 0)$. The values along an axis typically increase or decrease by 1. Here's an example of a two-dimensional coordinate system:

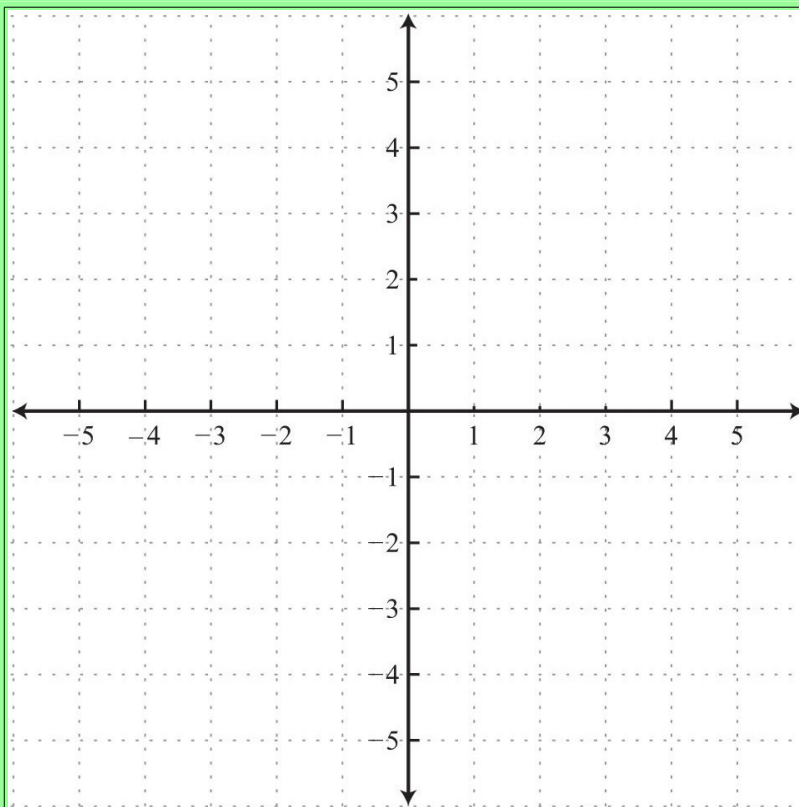


When specifying points on the coordinate system, we typically list the X-component first, followed by the Y-component. That is, the point $(2, 3)$, for example, has an X-value of 2 and a Y-value of 3. We could plot this point as follows:



Activity 1: Plotting points

Can you place these five points in their proper positions on the coordinate system below: $(3, 5)$, $(5, -2)$, $(1, 2)$, $(-3, -5)$, and $(-4, 0)$?



Another benefit of using the coordinate system for describing the position of different points is that it allows us to easily calculate and plot a point in the middle of two points. We call such a point the midpoint of the two points. If two points were drawn on a plain piece of paper, and we were asked to mark the point in the exact middle of those two points, it would be a long and (many times) inaccurate process. However, having the coordinates of the two points makes this simple. The midpoint of two points is calculated by adding the corresponding coordinates and dividing by two to find the average of each component. For example, the midpoint of the points (5, -2) and (1, 2) is given by the following coordinates:

$$\left(\frac{5+1}{2}, \frac{-2+2}{2} \right) = (3, 0)$$

That is, the x- and y-components of each point are averaged to find the x- and y-components of the midpoint. Similarly, the midpoint of the points (-3, -5) and (7, 3) is given by the following coordinates:

$$\left(\frac{-3+7}{2}, \frac{-5+3}{2} \right) = (2, -1)$$

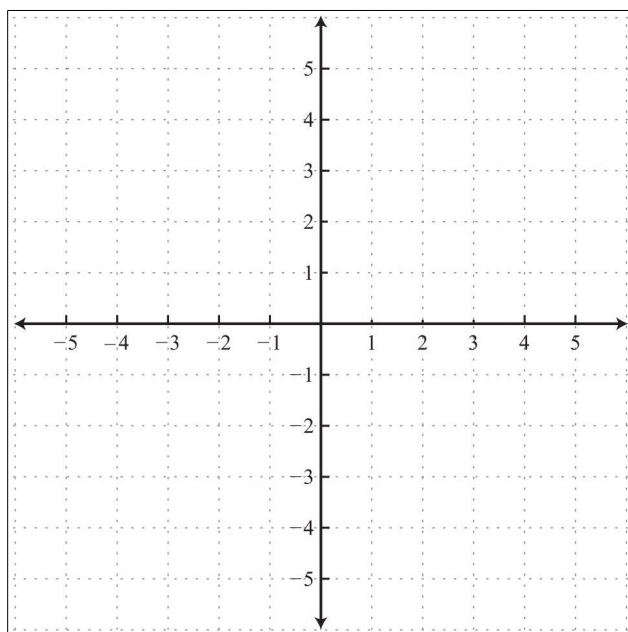
Try to calculate the midpoints of the following point combinations in the space below:

Points (0, 0) and (4, 4):

Points (-3, 5) and (-1, -3):

Points (4, -4) and (-4, -4):

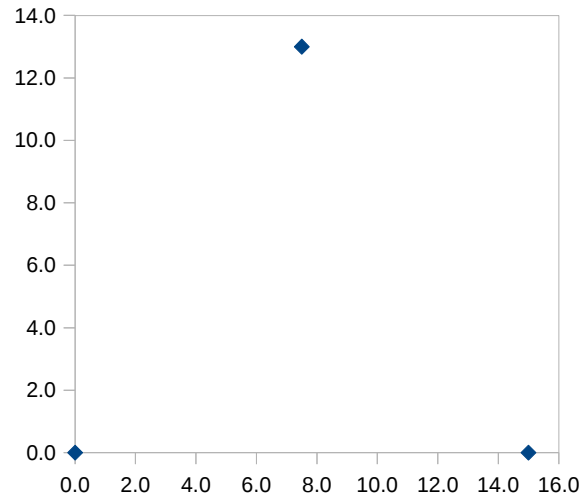
Now plot the given points and the calculated midpoints below:



The chaos game

The chaos game is an interesting (and, hopefully, fun) game that illustrates how seemingly random and chaotic things can produce something orderly (and beautiful) over time. It works in a simple manner. First, plot the three vertices of an equilateral triangle. An equilateral triangle is unique in that each of its three sides are the same length, and each of its three vertices are equidistant to each other. Since the internal angles of a triangle always sum to 180 degrees, each angle of an equilateral triangle is the same (i.e., each angle is 60 degrees).

Here is an example of the vertices of an (almost) equilateral triangle:



Notice that the three vertices of the equilateral triangle are drawn on a coordinate system at the points (0,0), (15,0) and (7.5,13). Although these points are not “perfectly” equidistant, they are nearly so (and good enough for this example).

Second, randomly select two of the three vertices. Suppose that the top and bottom-right vertices are randomly selected. Calculate the midpoint between these two vertices using the midpoint formula as illustrated above. In general, given two points (x_1, y_1) and (x_2, y_2) , the midpoint can be calculated as follows:

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)$$

The midpoint of the top and bottom-right vertices at points (7.5, 13) and (15, 0) is calculated as follows:

$$\left(\frac{7.5+15}{2}, \frac{13+0}{2} \right) = \left(\frac{22.5}{2}, \frac{13}{2} \right) = (11.25, 6.5)$$

The midpoint, (11.25, 6.5), is then plotted on the coordinate system.

The game now follows a repeatable pattern. The calculated midpoint at point (11.25, 6.5) is selected, along with one randomly selected vertex of the equilateral triangle. Suppose that the top vertex at point (7.5, 13) is randomly selected. A new midpoint is now calculated:

$$\left(\frac{11.25+7.5}{2}, \frac{6.5+13}{2} \right) = \left(\frac{18.75}{2}, \frac{19.5}{2} \right) = (9.375, 9.75)$$

this new midpoint is then plotted on the coordinate system, and the process continues (i.e., the latest midpoint is selected, along with a randomly selected vertex of the equilateral triangle). Here is the algorithm in pseudocode:

```

v1, v2, v3 ← three vertices of an equilateral triangle
plot v1, v2, and v3
p1, p2 ← two randomly selected vertices from v1, v2, and v3
m ← the midpoint of p1 and p2
plot m
repeat
    v ← a randomly selected vertex from v1, v2, and v3
    m' ← the midpoint of m and v
    plot m'
    m ← m'
until midpoints plotted >= 1500 or tired

```

By playing the chaos game for some time, what do you think will happen? Where do you think the points will focus on the coordinate system? Will they collect in any one place at all?

Activity 2: The chaos game...manually

Let's play this game a little bit to see if anything can be gleaned (i.e., if anything *reveals* itself). Mainly, we want to see if the points just randomly fill the coordinate system, or if they somehow produce something *orderly*.

To obtain results quickly, let's use blank sheets of paper (or even better, transparencies). Everyone should plot the exact same vertices of an equilateral triangle. From there, play the chaos game and plot approximate midpoints. That is, estimate as best as possible (by looking) where the midpoint should be plotted. Remember that, to plot the first midpoint, two random vertices are selected. From that point on, the last plotted midpoint and one randomly selected vertex is used to compute the next midpoint.

Each student should repeat this process on his/her own individual paper or transparency at least 25 times. It's best to plot at least 100 midpoints. The paper or transparency can then be combined with those of the other students to see what (if anything) is formed. If using transparencies, simply lay them on top of each other, aligning the vertices. The result should be clearly visible. If using paper, the procedure is the same; however, you will need to hold the stack of paper to the light (note that it may be difficult to see all of the points).

So what is the result?

Did you know?

For any pattern to be revealed, a single person would have to plot thousands of points (which would take a really long time). There is a whole branch of computer science that deals with making computers faster, particularly for large tasks. One of the techniques that addresses this involves dividing tasks so that small sub-tasks are done concurrently (i.e., at the same time). This field is called **high performance computing**, and the technique employed is called **parallelism**.

Activity 3: The chaos game...automated

As you can see, it takes a lot of points to plot anything reasonable. That is, in order to actually see if anything orderly is produced when playing the chaos game, a lot of midpoints must be calculated and plotted.

There are also a lot of other interesting things that can be tried. For example, what would happen if the three vertices were not equidistant; that is, what if the triangle were different? What if we plotted a square instead (i.e., four vertices instead of three)? What about a pentagon? What would happen if, instead of plotting midpoints, points at varying distances were plotted instead. For example, would the result be different if a three-fourths point was plotted instead each time (i.e., instead of 50% of the distance to the randomly selected vertex, 75% of the distance). What about 10% of the distance? What would happen if there was an additional rotation about the randomly selected vertex? That is, what would be the result if some midpoint (or other distance) were calculated and then rotated about the vertex some number of degrees (some angle)?

These are all interesting questions that may result in absolute randomness (or perhaps not!). It would be quite difficult to try these manually. Why not automate this process? A frequent task of computing professionals is to automate things that tend to be repeated. To illustrate a number of changes to the chaos game in a configurable manner, a simple application has been created to test out all of the options.

The goal of this activity is to play around with this application to see how changes to the chaos game affect the outcome. Try your hand at it!

Fractals

By now, you should have noticed that the result of the original chaos game (i.e., three vertices of an equilateral and repeatedly plotting midpoints) results in a beautiful triangular shape. In fact, this shape has an interesting property: it has its pattern repeated over and over. It's as if we could zoom in forever and obtain the same pattern! This property is found in fractals. This particular fractal has a name: the Sierpinski Triangle.

A **fractal** is a geometric shape that can be (infinitely) broken down into similar parts. This means that it is made up of many parts that are just smaller versions of the whole thing. And these smaller parts are then made up of even smaller versions of the whole thing. And this goes on infinitely (well, basically). Of course, our eyes can't see this going on infinitely because things get too small.

Randomness and probability

In the chaos game, random vertices of an equilateral were repeatedly selected. What does *random* mean exactly? We have seen that there is sometimes order in chaos if you look long and close enough. That

being said, there are things that seemingly happen in a haphazard manner, have no predictable pattern, and are not predetermined. Such things are called random. Think of selecting one of the three vertices. We have no way of predicting which vertex will be selected. Any one could be selected at each iteration. Think of rolling a die. There is no way to predict which number will be rolled next (other than to say that it will be between 1 and 6 inclusive).

Even with random things, human beings have come up with a way of measuring the randomness, and that way is called probability. Probability is a way of expressing the knowledge that something will happen. If something is definitely going to happen (i.e., it's absolute certainty), it is said to have a probability of 1. If something is definitely **not** going to happen (i.e., it's an impossibility), it has a probability of 0. Of course, there are an infinite amount of values in between 0 and 1. That is, the probability that something could happen can be stated as a value between 0 and 1.

Formally, probability is defined as the ratio of the number of ways of achieving success to the total number of possible outcomes. For example, consider the flipping of a coin. There are two possibilities when flipping a coin: either heads or tails. So there are two possible outcomes. The probability of rolling heads when flipping a coin is then $1/2$. Why? The only way to roll heads is, well, to roll heads. That is, there is only one way to achieving success. Since there are two possible outcomes when flipping a coin, the probability of rolling heads is therefore $1/2$ (one way of achieving success out of two possible outcomes). Similarly, the probability of rolling tails when flipping a coin is also $1/2$.

What is the probability of picking the top vertex in the chaos game? There's only one way to do so out of three possible vertices. Therefore, the probability is $1/3$.

Let's go back to rolling a typical (six-sided) die. What is the probability that a three is rolled? Rolling a three represents the only way to achieve success. There are six possible rolls of the die. Therefore, the probability of rolling a three is $1/6$.

What is the probability that an even number is rolled? Well, how many ways are there of achieving success? That is, how many ways can an even number be rolled? Three (rolling a two, four, or six). The possible outcomes are, of course, rolling a one through six. So the probability of rolling an even number is $3/6 = 1/2$.

What is the probability that a number less than six is rolled? There are five ways to roll a number less than six. There are six possible rolls of a die. The probability is therefore $5/6$.

What is the probability of rolling two die and getting a total of 7? This one is a bit tricky! We'll discuss this later (but if you want to know, it's $1/6$).

Activity 4: Heads and tails...sort of

For this activity, we will repeatedly flip two coins simultaneously and record the results. The two coins can either both be heads, both be tails, or they can be different (i.e., one is heads and one is tails). So there are three possible outcomes to this game. The probability of rolling two heads is therefore $1/3$, two tails is also $1/3$, and one heads and one tail is also $1/3$.

To make this interesting, let's make it a game. If the coins come up both heads, all of the male students in the class get a point. If the coins come up both tails, all of the female students in the class get a point. If one coin comes up heads and one comes up tails, the prof gets a point. The group (or individual) with the most points at the end of some number of flips (say, 15) wins.

Record the results below:

<u>Male Students</u>	<u>Female Students</u>	<u>Prof</u>
----------------------	------------------------	-------------

Over time, you will find that the students lose more often. It appears that the heads and tails combination happens significantly more often than initially thought. In fact, the previously calculated probabilities (each $1/3$) seem incorrect.

The purpose of this activity is to show that perceived probabilities are often different from actual probabilities (i.e., humans are not always so good at estimating probabilities). Think of it like this: there is only one way of rolling two heads (both coins must come up heads). There is only one way of rolling two tails (both coins must come up tails). Contrary to what was previously assumed, there are two ways of rolling one heads and one tails: the first coin can be heads and the second coin can be tails, or the first coin can be tails and the second coin can be heads! In fact, there are a total of four possible outcomes. The rolling of two coins can then be shown in the following table:

<u>Coin 1</u>	<u>Coin 2</u>
heads	heads
heads	tails
tails	heads
tails	tails

The probability of rolling two heads is $1/4$ (one possible way of achieving success out of four possible outcomes as show above). The probability of rolling two tails is also $1/4$. But the probability of rolling a heads and a tails is actually $2/4 = 1/2$! The prof wins 50% of the time! Another way of saying this is that the prof wins twice as much as either group of students! Ka-ching!

When you realize how rigged this simple *game* is, you can begin to think about how even more rigged other games (such as slot machines, the lottery, etc) are...

Try to calculate the probability of rolling three coins and getting three heads. Or three tails. Or one heads and two tails. Or one tails and two heads. To assist you, list all of the possible combinations of rolling three coins in the table below:

Coin 1

Coin 2

Coin 3

Probability of rolling three heads:

Probability of rolling three tails:

Probability of rolling one heads and two tails:

Probability of rolling one tails and two heads:

Random number generators

In the last RPi activity (titled *My Binary Addiction...Reloaded*), you may have noticed the use of a library that allowed the generation of random numbers. In the activity, it was used to generate random bits (0 or 1) for each of the two 8-bit numbers (in order to generate two random 8-bit numbers). The library was called `random` and was imported as follows:

```
from random import randint
```

The function `randint` was then used to generate the random numbers. As mentioned earlier in this lesson, a truly random event has no predictable pattern. Unfortunately, there is no easy way for that to happen using a computer, even for a task as simple as selecting random numbers within a specific range. As a solution, computers typically use a pseudo-random process to create random numbers in a range.

Definition: *A pseudo-random process is one that appears to be random but is technically deterministic in nature. This means that the process looks completely random, and yet its pattern is predictable and can be reproduced exactly.*

When tasked with creating a random number (or group of numbers), computers typically use a pseudo-random process to produce that number. This means that, given enough generated numbers, one can observe that the created numbers follow a certain sequence. For most purposes, however, the amount of numbers one would require to observe that the pattern is predictable is too large (so the numbers seem random).

For most cases, pseudo-random numbers are adequate and even beneficial. How? For instance, often researchers replicate experiments in order to confirm (or refute) experiments performed by peers. For a scientist to make any claim about any experiments carried out, enough information for someone else to reproduce those same results with an experiment of their own must be provided. When a random number (or list of random numbers) is used in an experiment, there would be no way of replicating those

exact numbers (and therefore obtaining the same results as in the original experiment). However, since computers produce pseudo-random numbers (which can be replicated exactly), it allows other researchers to carry out their own versions of the original experiment and crosscheck to see if the results are identical.

In order to actually generate repeatable (even predictable) patterns of *random* numbers, most pseudo-random number generators can be configured or initialized with a seed. A **seed** is just a number that is used to initialize a pseudo-random number generator. Most generators use the current time (to the second) as the seed. Clearly, this means that generating a sequence of random numbers will be different each time (since time moves forward continuously). We can, however, specify a seed of our own so that the pseudo-random number generator will always provide the same sequence of numbers!

Activity 5: Seeded pseudo-random numbers

Let's create a simple program generates 100 pseudo-random numbers (in the range 0-99) twice:

```
from random import randint

for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
print
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
```

Note the use of "\t" (which prints a *tab* and is typically eight spaces in width). Characters that begin with a backslash (\) are known as **escape characters**. Escape characters typically allow the representation of unprintable characters (such as a tab, newline, carriage return, etc). Most general purpose programming languages (including Python) support these unprintable characters. Here are some common ones:

```
\n    Linefeed (like pressing Enter)
\t    Horizontal tab
```

Note that there are many others, some of which are even programming language specific.

The program above first displays 100 random integers in the range 0-99, and formats them such that ten integers are displayed on each line (i.e., ten rows of ten columns). They are aligned at the columns via a horizontal tab that follows each integer. Note the comma at the end of the print statement. This instructs Python to omit a typically default linefeed at the end of the print statement. Once ten integers have been display on a single line (i.e., $i \% 10 == 0$), a blank line is added (via the solitary print statement). The program then displays another 100 random integers in the same manner. Note the variety (and randomness) of the generated integers, although generating 200 integers in the range 0-99 will undoubtedly produce duplicates.

In the above example, the pseudo-random number generator was seeded with the current time (by default since no numeric seed was provided). Let's modify the algorithm and use an identical (but specified) seed value for each 100 random integers:

```
from random import randint, seed

seed(123456)
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
print
seed(123456)
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
```

In this case, the numeric value 123456 is used as the seed to generate both groups of 100 integers. Technically, the seed could be any value (e.g., 0, some variable containing a value, even the mathematical constant pi). In fact, we could have randomly generated a seed!

```
seed(randint(0, 65535))
```

Now that you have seen how to use Python to generate pseudo-random integers, let's try to write a program that implements some of the activities shown earlier in this lesson.

Activity 6: Rolling two dice

Earlier we discussed rolling a single die and the associated probabilities of each possible roll. Then, we asked what the probability of rolling two dice and getting a total of 7. Let's try to write a Python program that iterates through all of the possibilities when rolling two dice:

```
dice1.py (~/.git/school/teaching/Winter2015-16/csc131/03 Chaos/code) - gedit
File Edit View Search Tools Documents Help
New Open Save Print Undo Redo Cut Copy Paste Find Replace
dice1.py x
# a list containing the frequencies of the 11 possible sums (2 through 12)
# this initializes a list of 11 elements, all with the value 0
dice_sums = [0] * 11

# display the possible rolls of two dice
print "Die1\tDie2\tSum"
# iterate through the values of die 1
for die1 in range(1, 7):
    # for each value of die 1, iterate through the values of die 2
    for die2 in range(1, 7):
        # calculate the sum of both dice
        dice_sum = die1 + die2
        # increment the frequency of each sum
        # the smallest possible sum, 2, is at index 0 of the list
        # so subtract 2 from the index
        # the frequency of rolling a 2 is at index 0 of the list
        dice_sums[dice_sum - 2] += 1

        # display the values for this roll
        print "{}\t{}\t{}".format(die1, die2, dice_sum)

# display the sum frequencies
print "\nSum\tFreq\tProb"
for i in range(len(dice_sums)):
    # i starts at 0, but we want to begin the sums at 2
    print "{}\t{}\t{}".format(i + 2, dice_sums[i], dice_sums[i] * 1.0 / sum(dice_sums))

Python Tab Width: 8 Ln 26, Col 15 INS
```

The comments in the source code above pretty well explain the statements. Note the last line of the program; specifically, the third parameter that replaces the formatting braces: `dice_sums[i] * 1.0 / sum(dice_sums)`. This expression takes the *i*-th sum frequency and converts it to a floating point number by multiplying it by 1.0. This product is then divided by the total number of sum frequencies (the function `sum()` calculates the sum of a list – in this case, the list `dice_sums`). The conversion of one of the terms to floating point is necessary to ultimately produce a floating point division.

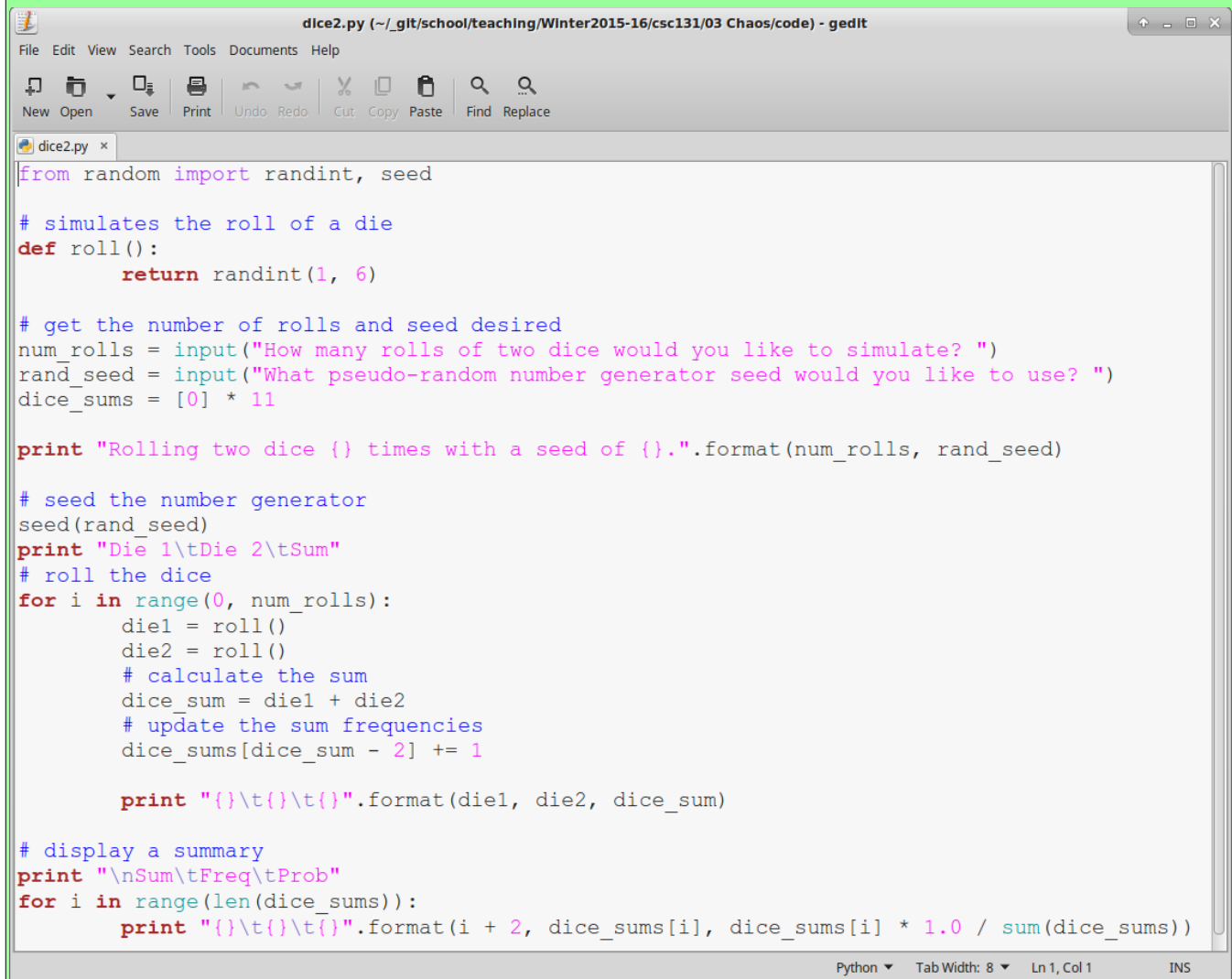
And here is the output of the code:

```
Terminal - jgourd@macchic
jgourd@macchio:~$ python dice1.py
Die1    Die2    Sum
1       1       2
1       2       3
1       3       4
1       4       5
1       5       6
1       6       7
2       1       3
2       2       4
2       3       5
2       4       6
2       5       7
2       6       8
3       1       4
3       2       5
3       3       6
3       4       7
3       5       8
3       6       9
4       1       5
4       2       6
4       3       7
4       4       8
4       5       9
4       6      10
5       1       6
5       2       7
5       3       8
5       4       9
5       5      10
5       6      11
6       1       7
6       2       8
6       3       9
6       4      10
6       5      11
6       6      12

Sum     Freq    Prob
2       1       0.0277777777778
3       2       0.0555555555556
4       3       0.0833333333333
5       4       0.1111111111111
6       5       0.1388888888889
7       6       0.1666666666667
8       5       0.1388888888889
9       4       0.1111111111111
10      3       0.0833333333333
11      2       0.0555555555556
12      1       0.0277777777778
jgourd@macchio:~$
```

The number of total sums possible is 36. We can calculate this by taking the sum of the frequencies (1 + 2 + 3 + 4 + 5 + 6 + 5 + 4 + 3 + 2 + 1 = 36). Since there are six rolls that sum to 7, then the probability of rolling two dice and getting a 7 is $6/36 = 1/6$. In fact, the probability of rolling two dice and getting a 7 is higher than anything else!

Now let's try to roll two dice many times and see what results we end up with. Perhaps the frequencies of the sums will match the frequencies shown above! Here's the code:



```
dice2.py (~/.git/school/teaching/Winter2015-16/csc131/03 Chaos/code) - gedit
File Edit View Search Tools Documents Help
New Open Save Print Undo Redo Cut Copy Paste Find Replace
dice2.py x
from random import randint, seed

# simulates the roll of a die
def roll():
    return randint(1, 6)

# get the number of rolls and seed desired
num_rolls = input("How many rolls of two dice would you like to simulate? ")
rand_seed = input("What pseudo-random number generator seed would you like to use? ")
dice_sums = [0] * 11

print "Rolling two dice {} times with a seed of {}".format(num_rolls, rand_seed)

# seed the number generator
seed(rand_seed)
print "Die 1\tDie 2\tSum"
# roll the dice
for i in range(0, num_rolls):
    die1 = roll()
    die2 = roll()
    # calculate the sum
    dice_sum = die1 + die2
    # update the sum frequencies
    dice_sums[dice_sum - 2] += 1

    print "{}\t{}\t{}".format(die1, die2, dice_sum)

# display a summary
print "\nSum\tFreq\tProb"
for i in range(len(dice_sums)):
    print "{}\t{}\t{}".format(i + 2, dice_sums[i], dice_sums[i] * 1.0 / sum(dice_sums))

Python Tab Width: 8 Ln 1, Col 1 INS
```

Again, the comments should be adequate to explain the source code. Much of the code is similar to the last program.

And here is the output (with 5,000 rolls):

```
Terminal - jgourd@macchio: ~
How many rolls of two dice would you like to simulate? 5000
What pseudo-random number generator seed would you like to use? 31337
Rolling two dice 5000 times with a seed of 31337.
Die 1  Die 2  Sum
5      2      7
6      1      7
5      2      7
4      6      10
6      2      8
3      1      4
2      3      5
4      6      10
5      3      8
2      1      3
3      2      5
6      1      7
5      2      7
1      6      7
3      5      8
3      5      8
1      4      5
3      5      8
6      3      9
5      6      11
1      2      3
1      4      5
2      6      8
1      6      7
2      4      6
...
2      4      6
6      2      8
3      6      9
5      4      9
3      2      5
2      2      4
4      1      5
6      5      11
4      4      8
1      6      7
Sum    Freq    Prob
2      126    0.0252
3      268    0.0536
4      407    0.0814
5      599    0.1198
6      712    0.1424
7      855    0.171
8      672    0.1344
9      533    0.1066
10     419    0.0838
11     273    0.0546
12     136    0.0272
jgourd@macchio:~$
```

Note the probabilities. Although they are not exactly the same as calculated before, they are very similar! More accurate results would be possible with a much higher number of rolls (try one million!).