(Python) Chapter 3: Repetition

3.1 while loop

Motivation

Using our current set of tools, repeating a simple statement many times is tedious. The only item we can currently repeat easily is printing the exact same message multiple times. For example,

```
print("I love programming in Python!\n"*10)
```

will produce the output:

```
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
```

Imagine that we wanted to number this list so that we printed:

```
1. I love programming in Python!
2. I love programming in Python!
3. I love programming in Python!
4. I love programming in Python!
5. I love programming in Python!
6. I love programming in Python!
7. I love programming in Python!
8. I love programming in Python!
9. I love programming in Python!
10. I love programming in Python!
```

Now, the times operator (*) is no longer capable of allowing us to produce this output. Luckily, Python provides us with multiple general tools for repetition where we'll simply specify which statements we want repeated and a way to determine how many times to repeat those statements.

Definition

The while loop is the most simple of the loops in Python. The syntax for the loop is as follows:

```
while <Boolean expression>:
    stmt1
    stmt2
    ...
    stmtn
stmtA
```

The manner in which this gets executed is as follows:

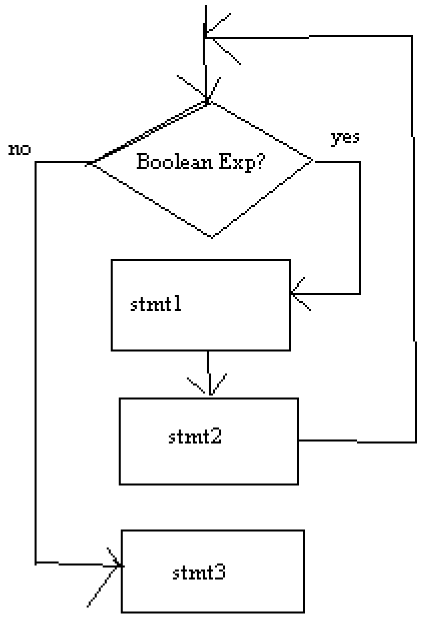
1) Evaluate the Boolean expression.
2) If it's true
   a) Go ahead and execute `stmt1` through `stmtn`, in order.
   b) Go back to step 1.
3) If the Boolean expression is false, skip over the loop and continue to `stmtA`.

The key to the repetition is that if the Boolean expression is true, after we complete the statement, we check the Boolean expression again instead of continuing. (In an if statement we would have continued on.)

Flow Chart Representation

Here is a flow chart representing of the following code segment:

```
while <Boolean expression>:
    stmt1
    stmt2
stmt3
```



I love C Programming 10 Times Over

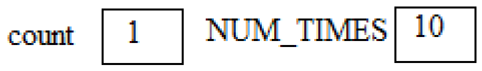
Now, with the while loop, we can more succinctly write code that prints out “I love programming in Python!” ten times in a numbered list.

Our first realization is that we must utilize the loop construct so that it can “count.” Namely, on any given running of the loop, we must have some way of “remembering” how many times we’ve already printed out message. In order to do this we’ll use an integer that will act as a counting variable. At any point in time, it’ll simply represent how many times we’ve printed our message.

Let’s take a look at a program that accomplishes our task:

```
def main():

    count = 1
    NUM_TIMES = 10

    while count <= NUM_TIMES:
        print(count,". I love programming in Python!", sep ="")
        count = count + 1

main()
```

At the very beginning of the program, our variables look like this:



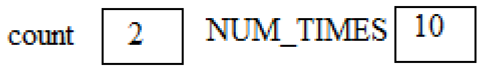
Thus, the while condition is true and the loop is entered. When we print, the value of count is 1, so we get the first line of output:

```
1. I love programming in Python!
```

Next, we execute the line

```
count = count + 1
```

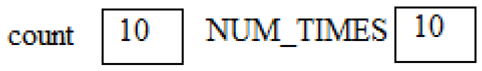
The left-hand side equals 2, thus, count will get set to two:



Since this a while loop and no more statements are indented, we go back to the top to check to see if the Boolean condition is true. Since count is less than or equal to NUM_TIMES, we enter the loop a second time, this time printing:

```
2. I love programming in Python!
```

Subsequently, we increment count to 3. Now, let's pick up execution when we get to the top of the loop and count is equal to 10:



Now, we enter the loop since the Boolean condition is still true. Then we print:

```
10. I love programming in Python!
```

Next, we increment count to 11. When we go back to the top of the loop, the Boolean condition is no longer true! Thus, we exit the loop after having printed out the desired output.

Alternatively, the following variation would work as well:

```
def main():

    count = 0
    NUM_TIMES = 10

    while count < NUM_TIMES:
        count = count + 1
        print(count,". I love programming in Python!", sep ="")

main()
```

It's important to realize that problems can be solved in different ways, even one as simple as this one. In our original solution, we used a 1-based counting system, starting our counting variable at 1. In the second example, we find if we change the order of statements in the loop, we must adjust the initial value of our count AND change our Boolean condition.

All of this is evidence that when dealing with loops, attention to detail and consistency are very important. It's fine to start count at either 0 or 1, but the rest of the code must be consistent with this starting value. (Note: in certain situations we'll find the 0-based counting system to be more natural and in others we'll prefer the 1-based system.)

In general, after examining this program in detail it should be fairly clear that we can execute any set of statements a set number of times using the same general construct:

```
count = 1

while count <= NUM_TIMES:
    # Insert code to be repeated here.
    count = count + 1
```

where NUM_TIMES is set to however many times the code block is to be repeated.

Using Loops to Print Out Different Patterns of Numbers

In the previous program, we used a loop variable to count from 1 to 10. Now, consider trying to count all the positive even numbers up to 100. One way to solve this problem would be to use a variable to count from 1 through 50, but then to multiply the variable by 2 when printing out the numbers, so instead of printing 1, 2, 3, ..., 50, the values 2, 4, 6, ..., 100 would get printed:

```
count = 1
while count <= 50:
    print(2*count)
    count = count + 1
```

Another technique would be to have count equal to the values that are being printed and then just print count. This means count must start at 2, and that we must add 2 to count each time around:

```
count = 2
while count <= 100:
    print(count)
    count = count + 2
```

Now, consider writing a code segment to print all the positive odd integer less than 100. We can employ techniques virtually identical to the ones above. In both cases, we must print a number 1 lower than what we were printing. Here are both solutions:

```
count = 1
while count <= 50:
    print(2*count-1)
    count = count + 1
```

```
count = 1
while count <= 100:
    print(count)
    count = count + 2
```

Finally consider writing a code segment to print the following sequence of numbers, one per line:

2, 4, 8, 16, …, 1024

In all of the previous examples, we would add a constant to the previous term to get the next term. In this exam, we need to multiply the previous term by 2 to get the next. This leads to the following solution:

```
count = 2
while count <= 1024:
    print(count)
    count = 2*count
```

Example: Add Up the Numbers from 1 to 100

Now let's look at a slightly more difficult example that utilizes the counting variable inside of the loop.

Consider adding the numbers 1, 2, 3, 4, …, 100. If you were to do this task by hand, you'd start a counter at 0, add in 1 to it to obtain 1, then add in 2 to it to obtain 3, then add in 3 to it to obtain 6, then add in 4 to it to obtain 10, etc.

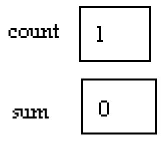
Let's use the while loop to automate this task.

We will need a counting variable similar to count in the previous example. But, we will also need a variable that keeps track of our running sum. Let's call this variable sum. Any variable that keeps a running tally going is known as an accumulator variable. The idea behind accumulator variables is critical. It is used in many practical programs.

Let's take a look at a program that solves this problem and trace through the first few steps carefully:

```
def main():

    MAX_TERM = 100

    sum = 0
    count = 1

    while count <= MAX_TERM:
        sum = sum + count;
        count = count + 1;

    print("The total is ",sum,".", sep="")

main()
```

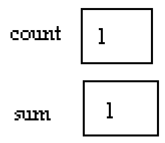
At the very beginning of the program, before the loop starts, memory looks like the following:



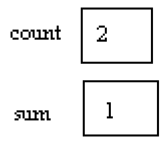
Initially, the Boolean expression for the while loop is true, since count, which is 1, is less than or equal to MAX_TERM. Now we encounter the critical line of code:

```
sum = sum + count;
```

Currently, sum is 0 and count is 1, so adding we get 1.This value is then stored back into the variable sum:



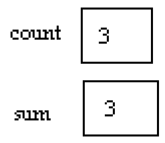
Next, we increase count by 1, so now our picture looks like:



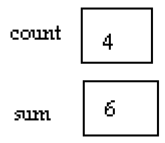
We then check the Boolean expression again and see that it is true. At this point in time, we've added in 1 to our total and are ready to add in 2. Now, we hit the line

```
sum = sum + count;
```

This time sum is 1 and count is 2, which add up to 3. This value gets stored in sum. Then, we follow this line by adding one to count, which changes to 3:



Now, we've added up 1 and 2 and are waiting to add in the next number, 3. The Boolean expression is still true, so we then add sum and count to obtain 6 and then change sum to 6. This is followed by adding one to count, making it 4:

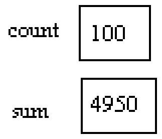


At this point, hopefully the pattern can be ascertained. At the end of each loop interation, sum represents the sum of all the numbers from 1 to count-1, and count is the next value to add into the sum. The key idea behind an accumulator variable is that you must initialize it to 0, and then each time you want to add something into it, you use a line with the following format:

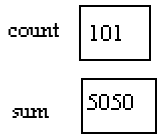
```
<accum var> = <accum var> + <expr to add in>;
```

Whichever variable is the accumulator variable is the one that is set, using an assignment statement. It's set to the old value of the variable plus whatever value needs to be added in.

In our example, if right before the very last loop iteration, the state of the variables is as follows:



Now, we go and check the Boolean expression and see that count is still less than or equal to MAX_TIMES (both are 100), so we enter the loop one last time. We add sum and count to get 5050 and then update count to be 101:



Now, when we check the Boolean expression, it’s false. We exit the loop and print sum.
To better understand this example, add the line:

```
print("count =",count,"sum =", sum)
```

as the last line of the body of the loop and adjust MAX_TERM as necessary:

```
    while count <= MAX_TERM:
        sum = sum + count;
        count = count + 1;
        print("count =",count,"sum =", sum)
```

When programs aren’t working properly, inserting simple print statements like this one can help uncover the problem. This process of finding mistakes is called debugging. Though inserting print statements isn’t the ideal way to debug, it’s the most simple way and recommended for beginners. More will be mentioned about debugging later. In this particular example, the print is to aid understanding, so that we can see what each variable is equal to at the end of each loop iteration.

Now, consider editing this program so it calculated the sum of the numbers 1, 3, 5, …, 99. We just change MAX_TERM to 99 and change our loop as follows:

```
    while count <= MAX_TERM:
        sum = sum + count
        count = count + 2
```

The key difference here is that count no longer represents how many times the loop has run. Rather, count simply represents the next number to add.

Yet another approach edits the loop as follows:

```
while 2*count-1 <= MAX_TERM:
    sum = sum + (2*count - 1)
    count = count + 1
```

In this approach, count keeps track of how many times the loop has run, but when we need to access the current term, we use an expression in terms of count (2*count – 1) that equals it.

As previously mentioned, programming problems can be solved in many different ways. It's important to get comfortable manipulating counting variables in loops as shown in these examples so that a programmer can solve different types of problems.

Sentinel Controlled Loop

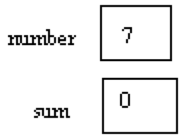
Each of the loops we've seen so far is a counting loop. Namely, a variable is used to keep track of how many times the loop has run, and the loop is terminated once it has run a particular number of times.

Now consider a slightly different problem: adding up a list on numbers until 0 is entered. In this problem, 0 is considered the sentinel value, since it represents the trigger for ending the loop. (The loop exits not because it's run a particular number of times, but because some value, known as the sentinel value, has been triggered.)

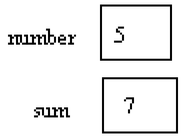
Just like the previous problem, we'll require the use of an accumulator variable, sum. We will also need to use a second variable to read in each number. Let's call this number. Now, we must read in each value into the variable number until this variable equals 0 (which means our loop runs so long as number doesn't equal 0.) Let's take a look at the program:

```
def main():

    sum = 0
    number = int(input("Enter your list, ending with 0.\n"))

    while number != 0:
        sum = sum + number
        number = int(input(""))

    print("The sum of your numbers is ",sum,".", sep="")

main()
```

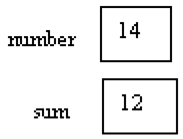
Let's trace through the example where the user enters 7, 5, 14, and 0. After the very first input, the state of the variables is as follows:



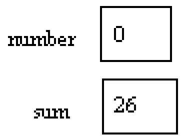
Since number isn't 0, the loop is entered. Adding the current values of sum and number yields 7, and we set sum to 7. Then, we read in the next number, 5:



Once again, number isn't 0, so we enter the loop again. Adding sum and number yields 12, and we set sum to 12. Then we read in 14 to number:



Since number isn't 0, the loop is entered again and sum and number are added to obtain 26, which is stored in sum and the next value read into number is 0:



Now, when we check the Boolean condition, it is evaluated as false and the loop is exited. Then, the value of sum, 26, is printed.

Guessing Game Example

A common game many young kids play is guessing a number from 1 to 100. After each guess, the person who chose the secret number tells the guesser whether their guess was too low or too high. The game continues until the guesser gets the number. The natural goal of the game is to minimize the number of guesses to get the number.

In this example, the "computer" will play the role of the person generating the secret number and user will be the guesser. We'll simply output the number of guesses it took to get the number.

Just like the previous example, since we don't know how many times the loop will run, we will check for a trigger condition to exit the loop. In particular, so long as the user's guess doesn't equal the secret number, our loop must continue.

A variable will be needed to store both the secret number and the user's guess. The rand function discussed in the previous chapter will be used to help generate the random number.

The basic strategy is as follows:

1) Generate a secret number
2) Set the number of guesses the user has made
3) Loop until the user's guess equals the secret number
    a) Prompt the user for a guess
    b) Read in a guess
    c) Output an appropriate message
    d) Update the guess counter
4) Output the number of moves the user took to guess the secret number.

Here's the program:

```
import random

def main():

    MAX_VAL = 100

    num_guesses = 0
    guess = -1

    secret = 1 + random,randint(1, MAX_VAL)

    while guess != secret:

        guess = int(input("Enter your guess.(1 to "+ str(MAX_VAL)+")\n"))

        if guess == secret:
            print("You win!")
        elif guess < secret:
            print("Your guess is too low. Try again!")
        else:
            print("Your guess is too high. Try again!")

        num_guesses = num_guesses + 1;

    print("You won with",num_guesses,"guesses.")

main()
```

This is the first example with an if statement inside of a loop. Since an if statement is a regular statement, it's perfectly fine to place in a loop. In this particular example, we are guaranteed to initially enter the loop because guess is set to -1, which is guaranteed not to equal the secret number, which is guaranteed to be in between 1 and 100.

When we read in the first guess, we need to decide which message to output the user. There are three possibilities, so we use a nested if to output the appropriate message. Finally, we use num_guesses as a counting variable through the loop. When the user enters the correct guess, "You win!" will be printed, the total number of guesses will be updated and when the Boolean expression is checked again, the loop will exit and the correct number of total guesses will be printed.

Example: Guessing Game Rewritten using a flag-controlled loop

Yet another technique often used to control a loop is a flag controlled loop. Simply put, a flag is a variable that keeps track of whether or not to enter a loop. A common method for using a flag is to set it to 0 while the task is not done, to indicate that the loop should continue and then set it to 1 when the task is done. This general technique can be used in many settings.

Here is the guessing game rewritten to use a flag to control the loop. A couple other changes have been made to streamline this version:

```
import random

def main():

    MAX_VAL = 100

    num_guesses = 0
    guess = -1
    done = False

    secret = 1 + random.randint(1, MAX_VAL)

    while not done:

        guess = int(input("Enter your guess.(1 to "+ str(MAX_VAL)+")\n"))

        if guess == secret:
            done = True
        elif guess < secret:
            print("Your guess is too low. Try again!")
        else:
            print("Your guess is too high. Try again!")

        num_guesses = num_guesses + 1;

    print("You won with",num_guesses,"guesses.")

main()
```

The key here is when we realize that the correct guess was made, we must set our flag done to True, to indicate that the loop is now done. This technique of using a Boolean variable to control a loop running is very common.

Idea of a Simulation

Now that we can repeat statements many times and generate random numbers, we have the power to run basic simulations. One practical use of computers is that they can be used to simulate items from real life. A nice property of a simulation is that it can be done really quickly and often times is very cheap to do. Actually rolling dice a million times would take a great deal of time, and actually running a military training exercise may be costly in terms of equipment, for example. As you learn more programming tools, you'll be able to simulate more complicated behavior to find out answers to more questions. In this section we'll run a simple simulation from the game Monopoly.

There are 40 squares on a Monopoly board and whenever a player rolls doubles, they get to go again. If they roll doubles three times in a row, they go to jail. There are a few more ways that affect movement, but these are more complicated to simulate. Let's say we wanted to know how many turns it would take on average to get around the board. We have the tools to simulate this. One simplification we'll make is that we'll stop a turn at three doubles in a row and we won't put the player in jail. We are making this simplification so that our code will be manageable. We will still get a reasonable result without having to add complicated code. Whenever writing simulations, we must make decisions about how accurately we want to carry out our simulations. If an extremely accurate answer is necessary, we must spend more time to make sure that the details of our simulation match reality. But, if we are running our simulation just to get a ballpark figure, as is the case in this simulation, certain simplifications are warranted. (It's relatively rare for us to roll three doubles in a row, so if our model is inaccurate in this one instance, it won't affect our overall results much. Technically, we'll roll three doubles in a row once every 216 rolls or so, which is less than 1% of the time.)

For our simulation, we want to run several trials, so we'll create an outside loop that loops through each trial. In a single trial we will accumulate rolls of the dice until the sum gets to 40. As we are running the trial, we will count each turn. After the first turn that exceeds a sum of 40 for all the dice rolls in that trial, we stop the trial and count the number of turns it took us to "get around the board." We add this to a variable and will use this sum to calculate our end average. Here is the program in full:

```python
# Arup Guha
# 7/6/2012
# Simulates how many turns it takes to go around a Monopoly board
# (without special squares...)
import random

def main():

    # Initialize necessary variables.
    i = 0
    NUMTRIALS = 100000
    sumturns = 0

    # Run each trial
    while i < NUMTRIALS:

        # Set up one trip around the board.
        spot = 0
        turns = 0

        # Stop when we get around the board.
        while spot < 40:

            # Start this turn.
            turns = turns + 1

            # Make sure we don't go more than three times.
            doublecnt = 0
            while doublecnt < 3:

                # Do this roll and update.
                roll1 = random.randint(1,6)
                roll2 = random.randint(1,6)
                roll = roll1 + roll2
                spot = spot + roll

                # Get out if we didn't get doubles.
                if roll1 != roll2:
                    break

                doublecnt = doublecnt + 1

        # Update number of total turns.
        sumturns = sumturns + turns
        i = i+1

    # Print out our final average.
    average = sumturns/NUMTRIALS
    print("Average is",average)

main()
```

Note: When we run this code, it takes several seconds to complete, since so much work is being done in the simulation. Also, it's important to note that python, due to its interpreted nature, runs slower than other languages. A comparable simulation in C takes less than a second.

Though this program is more complex that ones we've previously seen, if you take it apart line by line, there is nothing complicated in its logic. The key is to break down each subtask into logical steps and then convert those steps into code. It this program, we have three levels of loops. The outer most level takes care of multiple trials while the middle loop takes care of a single trial and the inner most loop takes care of a single turn, which can have multiple dice rolls, due to doubles. In regular Monopoly, when someone rolls doubles three times in a row, they go to square number 10, or jail. From there, they lose a turn and can get start rolling again, if they pay $50. If we ignore the monetary consequence, we can simulate this relatively easily, by checking to see if doublecnt is 3, resetting spot to 10 in that case, and adding an extra one to turns.

A great way to improve your programming skills is to think about some event from real life and see if you can write a simulation for it. In this program, we found out that on average, it takes about 5.3 to 5.4 turns to circle the Monopoly board!

3.2 for loop

Counting Loop

A while loop is a general purpose loop that can run either a specified number of times using a counting variable, or an unknown number of times, if it's controlled by some other condition. If the number of loop iterations is known, then a while loop is sometimes clunky, because the information needed to determine how many times it will run is located in three separate places in the code - where the counting variable is initialized, where the Boolean condition is checked, and where the counting variable is incremented.

The for loop provides a syntax where all of this information is provided within the same line, so that a programmer can easily see exactly how the loop will be controlled without scrolling all over the code.

The basic syntax of a for loop is as follows:

```
for <variable> in <range>:
    stmt1
    stmt2
    ...
    stmtn
```

A range can be specified in three different ways. The most simple way to specify a range is to provide a single number. For example, `range(10),` represents the set of integers {0, 1, 2, ..., 9}. In general, `range(n),` represents the set of integers {0, 1, 2, ..., n-1}.

The way this loop works is as follows:

The variable is set to the first number specified in the range. Then, the statements stmt1 through stmtn are executed. Next, the variable is set to the second number in the range and the statements stmt1 through stmtn are executed again. The loop continues in this manner until the variable has been set to each number in the range, in order, and all the statements in the loop have been run each time. Here is the most simple for loop:

```
for i in range(10):
    print("I love programming in Python!")
```

If we wanted a numbered list starting at 1, we could do the following:

```
for i in range(10):
    print((i+1),". I love programming in Python!", sep="")
```

Now, we can express our sum program succinctly as follows:

```
def main():

    MAX_TERM = 100

    sum = 0
    for count in range(MAX_TERM):
        sum = sum + (count + 1)

    print("The total is ",sum,".", sep="")

main()
```

All the key logic is the same here as the while loop version, but the code is more succinct, since the counting information is represented all on the same line, more compactly.

Obviously, it is a bit limiting that so far we have been forced to start all of our loops at 0. But, we can specify ranges that don't start at 0 by adding a second parameter. For example, the expression `range(a,b)`, represents the set of numbers {a, a+1, a+2, ..., b-1}. Thus, `range(2, 6)` represents the set of values {2, 3, 4, 5}. Notice that the first value is inclusive, meaning it's included in the range and the second value is exclusive, meaning it's not included in the range. While this seems strange, it's consistent with our intuitive notion that the number of values represented is `b - a`. In the given example, we have 6 - 2 = 4, and there are exactly 4 values in the set {2, 3, 4, 5}, but 5 values in the set {2, 3, 4, 5, 6}. Note that this specification is inconsistent with the specification of the `random.randint` method which allows both endpoints to be generated as random numbers. With this added flexibility in the range function, we can now write our sum program a bit more naturally, as follows:

```
def main():

    MAX_TERM = 100
    sum = 0
    for count in range(1, MAX_TERM+1):
        sum = sum + count

    print("The total is ",sum,".", sep="")

main()
```

Finally, consider the situation where we want to add the odd integers from 1 to 99, inclusive. In our previous solution, we simply added 2 to our counting variable, each time through the loop. The range function allows us to specify integers that are equally spaced out as well with the addition of a third parameter.

The function call `range(a, b, c)` specifies the numbers starting at a, with a step size of c, that are less than b. The step size simply specifies what to add to get to the next number. For example, `range(12, 30, 4)` represents the set {12, 16, 20, 24, 28} and `range(30, 40, 2)` represents the set {30, 32, 34, 36, 38}. Using this incarnation of the range function, we can very easily add the odd numbers less than 100:

```
def main():

    MAX_TERM = 100
    sum = 0
    for count in range(1, MAX_TERM, 2):
        sum = sum + count

    print("The total is ",sum,".", sep="")

main()
```

Now that we have specified the most general range function, we can write a short program that allows the user to enter these three parameters in their sum:

```
def main():

    sum = 0
    start = int(input("Please the starting integer."))
    end = int(input("Please the end integer."))
    step = int(input("What is your step?"))

    for i in list(range(start,end+1,step)):
        sum = sum + i

    print("Total is",sum)

main()
```

Note that because of how the range function works and how users will intuitively understand a range, we can adapt bu simply passing in end+1 to the range function, to ensure that the value end will be included in the sum.

Diamond example

Consider the problem of printing out a diamond. Let's define diamonds for the positive odd integers, where the integer repesents the number of rows and the number of stars on tin the middle row. Here are diamonds of size 1, 3, 5 and 7:

```
*              *               *               *
              ***             ***             ***
               *             *****           *****
                              ***           *******
                               *             *****
                                              ***
                                               *
```

The key here is that we realize that there isn't one single pattern in play. The first "half" of the design follows a simple pattern while the second half of the design follows a different pattern. Thus, we should separate our work into two loops. For example, if n is 7, we want our first loop to print the first four lines of the design and our second loop to print the last three lines of the design.

We further notice that the number of spaces decreases by 1 for the first half of the design and increases by 1 in the second half of the design. Secondly, the number of stars increases by 2 in the first half of the design and decreases by 2 in the second half of the design.

Though there are many ways to write this program, we'll use our for loop counting variable to represent the number of spaces on each line. Thus, the first loop will "count down" while the second loop will "count up." Also, we could create a formula for the number of stars on the basis of the number of spaces, but it's easier to just create a separate variable that stores the number of stars and update that each time in our loop accordingly.

In addition, we'll add a minimal bit of error checking in our program. If the user enters an even number, we'll simply tell them that we only make odd sized diamonds.

Note that we use integer division to ensure the accurate answers for the number of spaces and stars, since the range function needs integers to operate properly. Also note that the step in a range can be negative. This is useful for us since we want to go through the space values in backwards order for the top half of the design. When n = 7, we want our first line to have 3 spaces, our second line to have 2 spaces, our third line to have one space and our fourth line to have zero spaces.

A good exercise to solidify your understanding of loops would be to write this program without using the '*' operator for repeatedly printing out a string.

```
# Arup Guha
# 6/27/2012
# Prints a diamond of star characters.

def main():

    carats = int(input("How many carats is your diamond?\n"))

    if carats%2 == 0:
        print("Sorry, we only make odd carat diamonds.")
    else:

        spaces = carats//2
        stars = 1

        # Print top part of diamond here
        for spaces in range(carats//2, -1, -1):

            # Print out spaces number of spaces.
            print(" "*spaces + "*"*stars)
            stars = stars + 2

        # New starting values for rest of the sequence.
        stars = carats - 2

         # Print bottom part of the diamond.
        for spaces in range(1,carats//2+1):

            # Print out spaces number of spaces.
            print(" "*spaces + "*"*stars)
            stars = stars - 2

main()
```

<u>3.3 Loop Control Elements</u>

Motivation

Even with all the different loop elements we've learned, occasionally loops are difficult to design so that they properly execute what we would like. Two statements that will help us with tricky loop situations are break and continue.

Break

Occasionally, we might be in a loop but run into a situation where we immediately want to exit the loop, instead of waiting until the loop runs its course. This is what the break statement allows us to do. Whenever we are inside of a loop and we hit a break statement, that immediately transfers control to the point right after the end of the loop.

Obviously, since we don't ALWAYS want to be broken out of a loop, it's nearly always the case that a break statement appears inside of an if statement inside of a loop.

Alternate Loop Design Using Break

Some programmers don't like checking the loop condition in the beginning of the loop. Instead, they set up each of their loops as follows:

```
while True:
```

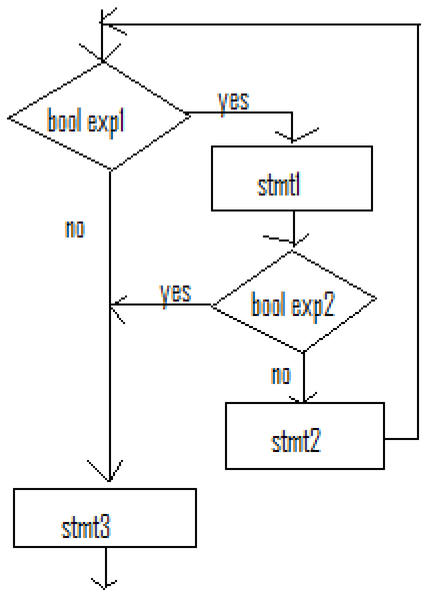
and use the break statement to get out of the loop whenever they want to. Similar to this:

```
while True:

    stmt1
    ...
    if <exit condition1>:
        break
    stmt2
    ...
    if <exit condition2>:
        break
    stmt3
    ...
```

Flow Chart Representation

As an example, consider the following segment of code and its flow chart representation:

```
while <bool expr1>:
    stmt1
    if <bool expr2>:
        break
    stmt2

stmt3
```



Prime Number Testing Example

A prime number is a number that is only divisible by 1 and itself. The standard method to test to see if a number is prime or not is to try dividing it by each number starting at 2, to see if there is any remainder or not. If any of these numbers divides evenly, meaning that it leaves no remainder, then the number is not prime.

In the following program, we will ask the user to enter a number and then determine whether it is prime or not. Once we find a number that divides evenly into the number our user has entered, there will be no further need to try other divisions.

```
def main():

    n = int(input("Please enter a number.\n"))

    # Try dividing this number by each possible divisor.
    isPrime = True
    for div in range(2, n):
        if n%div == 0:
            isPrime = False
            break

    # 2 is the smallest prime, so screen these out.
    if n < 2:
        isPrime = False

    # This number is prime, so print it.
    if isPrime:
        print(n,"is prime.")
    else:
        print(n,"is NOT prime.")

main()
```

The key idea here is that we try successively dividing n by 2, then 3, etc. If we ever find a number that divides evenly that is less than n, we immediately break. This means that instead of going back up to set div to the next value, we immediately skip to the next statement after the loop, the if statement. In essence, we stop the loop in its tracks when div equals the first value greater than one that divides evenly into n. If we never find such a value, the loop completes its course and isPrime is never set to false.

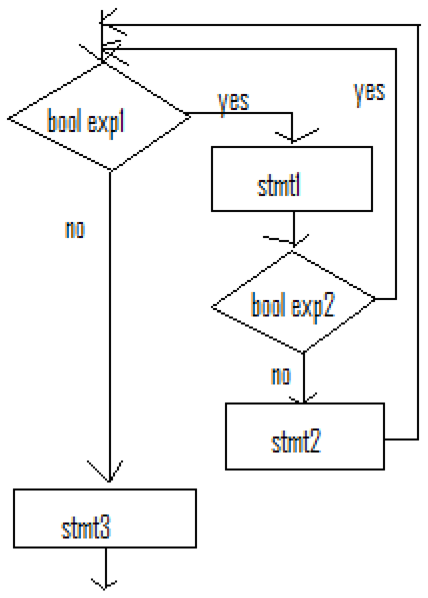
Continue

Continue is very similar to break except that instead of getting out of the loop completely, it simply skips over the rest of that following loop iteration and returns to the top of the loop. This may be desired if an input value can’t be processed and a new input value is needed.

Flow Chart Representation

As an example, consider the following segment of code and its flow chart representation:

```
while <bool expr1>:
    stmt1
    if <bool expr2>:
        continue
    stmt2

stmt3
```



Notice that only one tiny change had to be made in this flow chart as compared to the flow chart illustrating the use of break.

Reading in Valid Test Scores Example

In the following example we will read in the first *n* valid test scores from the user and calculate their sum. Whenever the user enters an invalid score, we'll simply ignore it. We will use an if statement to screen out invalid test scores and continue in these situations.

Here is the program:

```
def main():

    total = 0

    n = int(input("How many valid test score to enter?\n"))

    print("Enter your scores, one per line.")
    print("Invalid scores will be ignored.")
    count = 0

    while count < n:

        score = int(input(""))
        if score < 0 or score > 100:
            continue

        count = count + 1
        total = total + score;

    print("Your valid scores add to ",total,".", sep="")

main()
```

Here, any time we read in an invalid score, we DON'T process it. Namely, we go back up to the top of the loop so that we can start the process to get the next score, without counting this one.

3.4 Python Turtle

Logo Turtle

In the early 1980s, a programming language called Logo was created to get kids interested in programming. The language allowed the programmer to control a turtle that drew on a screen. The turtle held a pen, could move forward and could turn any number of degrees in either direction. The turtle could also pick its pen up or put it down. In this manner, many designs could be drawn. Python has implemented its own version of the turtle so that beginning programmers can see the fruits of their efforts in a visual display. Python's turtle offers more options than the original turtle, with the ability to use many colors, fill in shapes and much more. In this section, only a brief introduction to the turtle will be given. For a complete list of turtle commands, reference the following web page:

`http://docs.python.org/library/turtle.html`

Getting Started With the Turtle

In order to use the turtle, you must do the following import statement:

```
import turtle
```

An important reminder is that when we do this statement, we are allowed to use all of the code included for the python turtle, but since that file name is turtle.py, we can NOT name our file turtle.py. Thus, for all of your turtle programs, pick a file name DIFFERENT than turtle.py.

Each turtle function must be called with the following syntax:

```
turtle.function(<parameters>)
```

Here is a list of the most simple turtle functions:

```
penup() - Picks the turtle's pen up.

pendown() - Puts the turtle's pen down.

forward(n) - Moves the turtle forward n pixels.

right(n) - Turns the turtle's heading right by n degrees.

left(n) - Turns the turtle's heading left by n degrees.
```

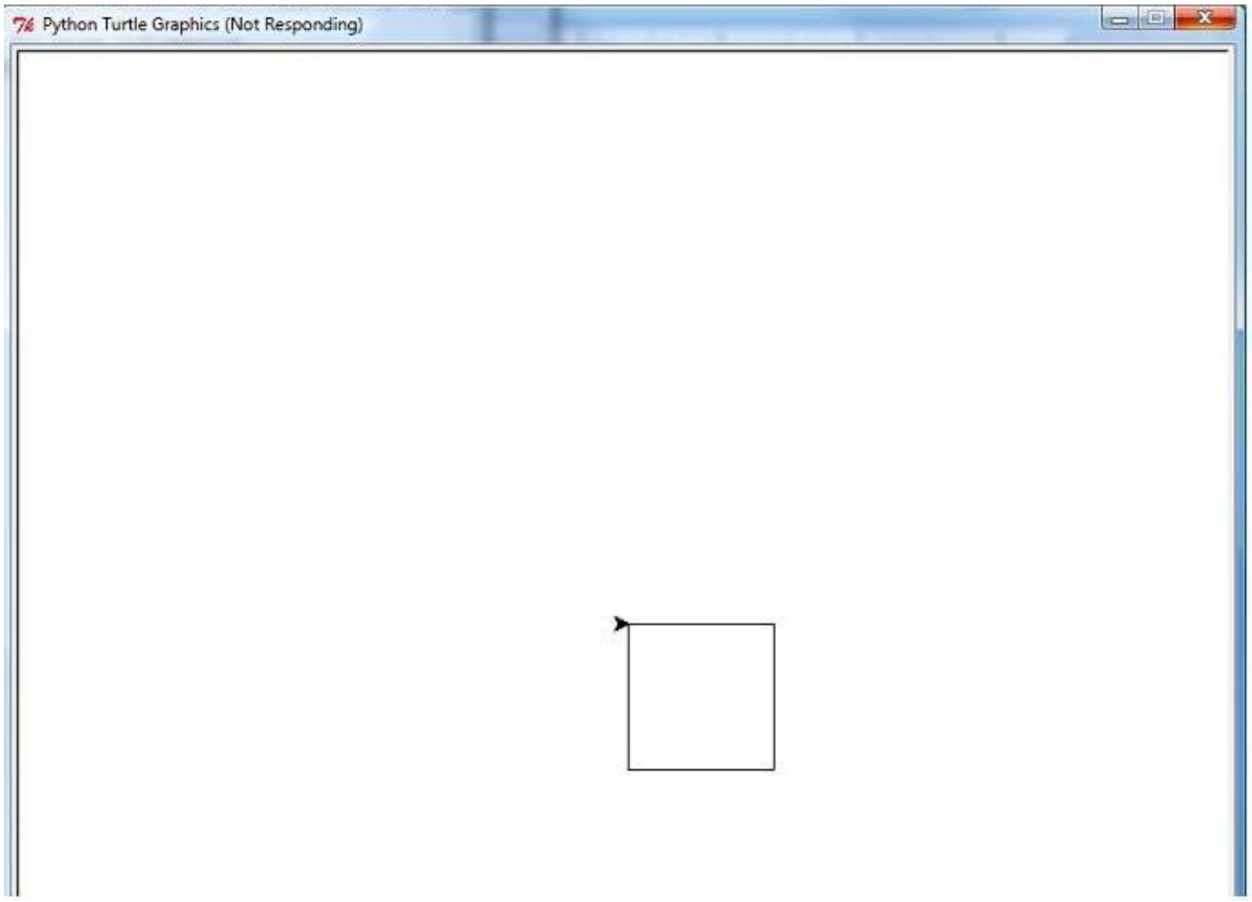
The easiest way to learn with the turtle is to try out examples. Let's look at four examples.

Square Example

A square is nothing but going forward and turning right, repeated 4 times. Of course, it would be just as easy to turn left each time. The standard turn, of course, is 90 degrees.

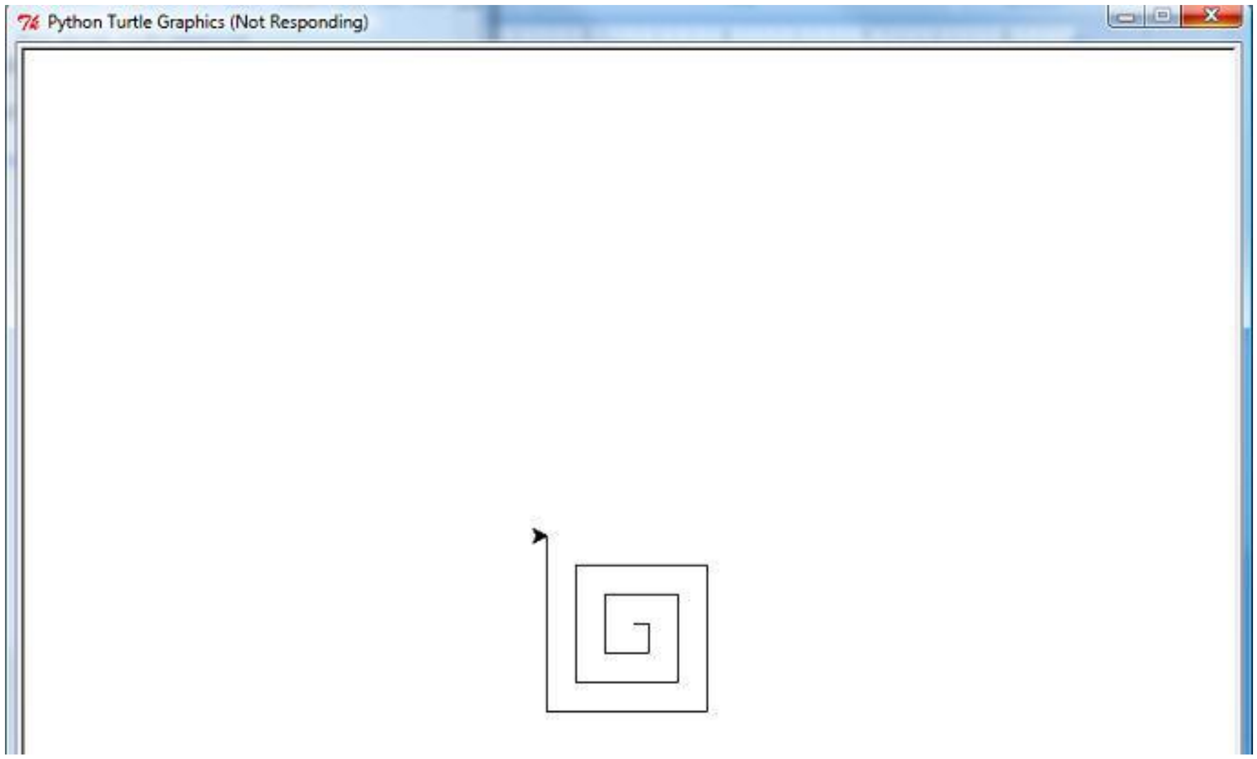
```
import turtle

def main():

    turtle.pendown()

    for cnt in range(4):
        turtle.forward(100)
        turtle.right(90)

main()
```

The result of this is as follows:



Spiral Square Example

We can make our design a bit more complicated by using the square idea, but by changing the length of each side, successively. Doing this will allow us to create a "spiral square" design:



The only change we need to make in our code is to change the length of the side, increasing it by 10 each time, each time through the loop.

```
import turtle

def main():

    side = 10
    n = int(input("How many sides do you want on your spiral square?\n"))

    turtle.pendown()
    for cnt in range(n):
        turtle.forward(side)
        turtle.right(90)
        side = side + 10

main()
```

Mountain Example

In the following program, we will allow the user to enter the number of mountains they want, and we'll print out a mountain range with that many mountains, all equal sized. To keep things simple, each mountain will be shaped like an upside down V and the slope both up and down the mountain will be 45 degrees. (This means that the angle formed by the peak of each mountain is 90 degrees.)

Since the turtle starts in the middle of the screen, we will first have to move to the left side of the screen and we will then put our pen down and draw our mountains from left to right. Based on the number of mountains the user wants, we will adjust the size of each mountain so that the mountains range from an x value of -300 to positive 300. In the program, we will have a variable called size that will actually be half of the length of the base on one triangle. Thus, if we have n triangles, each base will be length 600//n (since we need integers to specify pixel lengths) and side will be set to 300//n. Using this variable, we can calculate the length of each side of each mountain as $side\sqrt{2}$, since the slope of the mountain can be viewed as the hypotenuse of a 45^o-45^o-90^o triangle.

Here's the program:

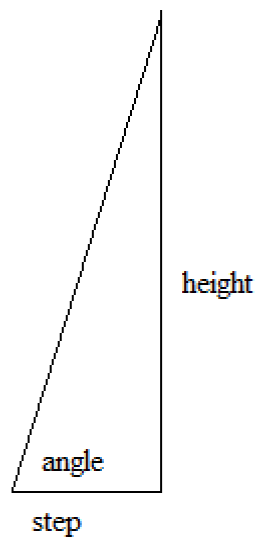
```
import turtle
import math

def main():

    n = int(input("How many mountains do you want?\n"))

    # Moves turtle to the left end of the screen.
    turtle.penup()
    turtle.right(180)
    turtle.forward(300)
    turtle.right(180)
    turtle.pendown()
    step = 300//n

    for cnt in range(n):

        # Use the Pythagorean theorem here, or the 90-45-45 triangle.
        sidelength = step*math.sqrt(2)

        # Turn to go up the mountain and then forward.
        turtle.left(45)
        turtle.forward(sidelength)

        # Turn to go back down and go down.
        turtle.right(90)
        turtle.forward(sidelength)
        turtle.left(45)

main()
```

More Advanced Mountain Example

Often times, students who enjoy programming don't understand the relevance of mathematics to programming. However, computer science sprouted out of mathematics departments in universities in the 1970s and 1980s and the connection between the two fields is intimate. Computers are helpful in automating solving problems, and invariably, some problems and patterns are mathematical in nature. In this example, we want to create mountains of a set height, but variable width. In order to do this properly, we'll have to employ some trigonometry.

In the previous example, the height of each mountain was directly related to the width, which meant that no matter what the user entered, the angles for all of the turns were the same. But, if the height and total width are fixed and the number of mountains are variable, then we have to make the appropriate angle calculation.

The key is to break down one mountain. Since each mountain is symmetrical, we really just need to look at one half of the triangle:

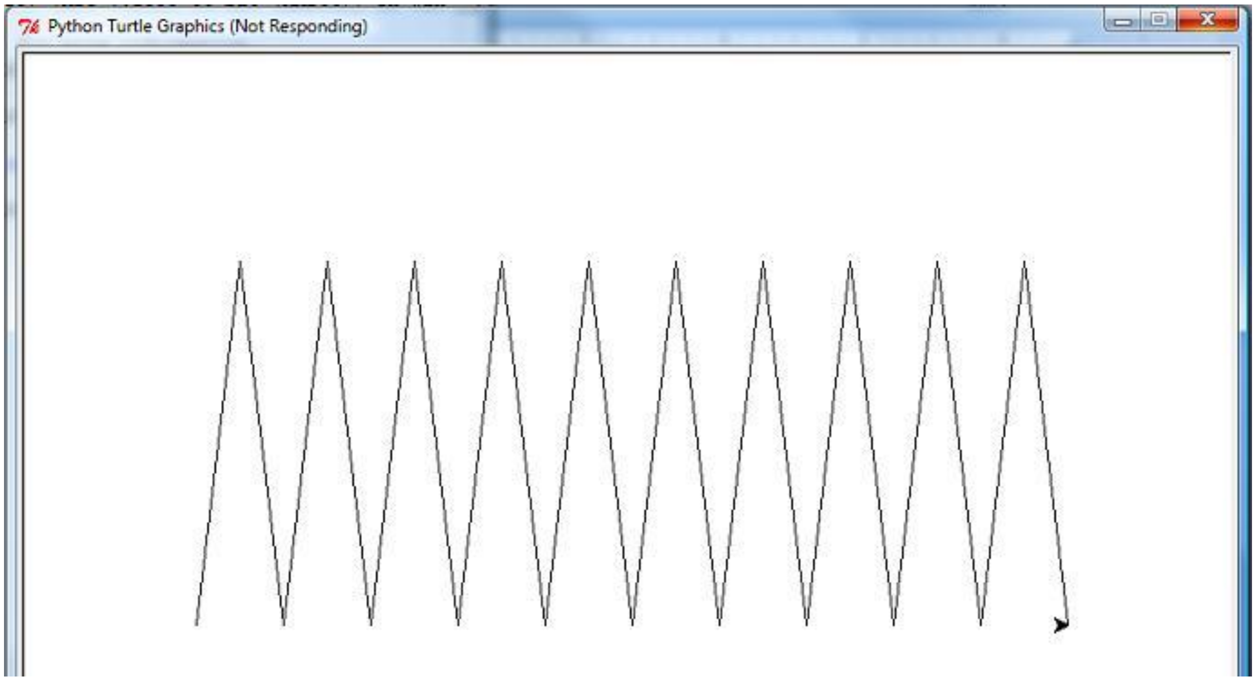


Using trigonmetric ratios, we find that $\tan(angle) = \frac{height}{step}$. Using the inverse tangent function, it follows that $angle = \operatorname{atan}\left(\frac{height}{step}\right)$. The peak angle in the picture is the complement of angle. The Pythagorean Theorem can be used to find the length of the side of the mountain, since the other two sides of the right triangle are known.

The program, in its entirety, is included on the next page.

```python
import turtle
import math

def main():

    n = int(input("How many mountains do you want?\n"))

    # Moves turtle to left end of the screen.
    turtle.penup()
    turtle.right(180)
    turtle.forward(300)
    turtle.right(180)
    turtle.pendown()

    # Setting our step size, for each side of the mountain.
    step = 300//n
    height = 250

    for cnt in range(n):

        # Get angle and convert to degrees.
        angle = math.atan(height/step)
        angle = angle*180/math.pi

        # Calculate our side length.
        side = math.sqrt(step**2 + height**2)

        # Now we can draw one mountain.
        turtle.left(angle)
        turtle.forward(side)
        turtle.right(2*angle)
        turtle.forward(side)
        turtle.left(angle)

main()
```

Here is the result of entering 10 for the number of mountains in this more advanced mountains program:



3.5 Practice Programs

1) Write a program that asks the user for a positive even integer input n, and the outputs the sum 2+4+6+8+...+n, the sum of all the positive even integers up to n.

2) Write a program to take in a positive integer n > 1 from the user and print out whether or not the number the number is a perfect number, an abundant number, or a deficient number. A perfect number is one where the sum of its proper divisors (the numbers that divide into it evenly not including itself) equals the number. An abundant number is one where this sum exceeds the number itself and a deficient number is one where this sum is less than the number itself. For example, 28 is perfect since 1 + 2 + 4 + 7 + 14 = 28, 12 is abundant because 1 + 2 + 3 + 4 + 6 = 16 and 16 is deficient because 1 + 2 + 4 + 8 = 15.

3) Write a program that allows a user to play a guessing game. Pick a random number in between 1 and 100, and then prompt the user for a guess. For their first guess, if it's not correct, respond to the user that their guess was "hot." For all subsequent guesses, respond that the user was "hot" if their new guess is strictly closer to the secret number than their old guess and respond with "cold", otherwise. Continue getting guesses from the user until the secret number has been picked.

4) Write a program that asks the user to enter two positive integers, the height and length of a parallelogram and prints a parallelogram of that size with stars to the screen. For example, if the height were 3 and the length were 6, the following would be printed:

```
******
 ******
  ******
```

or if the height was 4 and the length was 2, the following would be printed:

```
**
 **
  **
   **
```

5) Write a program that prints out all ordered triplets of integers (a,b,c) with a < b < c such that a+b+c = 15. (If you'd like, instead of the sum being 15, you can have the user enter an integer greater than or equal to 6.) You should print out one ordered triplet per line.

6) Write a program that prompts the user for a positive integer $n \leq 46$ and prints out the nth Fibonacci number. The Fibonacci numbers are defined as follows:

$$F_1 = 1, F_2 = 1 \text{ and } F_n = F_{n-1} + F_{n-2}.$$

The reason the input is limited to 46 is that the 47th Fibonacci number is too large to be stored in an integer. Your program should calculate the numbers the same way one would do by hand, by adding the last two numbers to get the following number in succession: 1+1 = 2, 1+2 = 3, 2+3 = 5, 3 + 5 = 8, etc.

7) Write a program using the turtle that draws a spiral triangle design, similar to the sprial square.

8) Write a program using the turtle that asks the user if they want a 5-pointed or 6-pointed star and draw the star. to extend the program, allow the user to enter any number 5 or greater and draw the corresponding star.

9) Write a program using the turtle that draws a track with several lanes. You may ask the user to enter an integer in between 1 and 8 for the number of lanes. A track typically consists of two straightaways with semicircles on both sides. A lane is enclosed between two of these shapes. Thus, a track with 6 lanes should have 7 figures of similar shape, enclosed in one another.

10) Write a program using the turtle that creates a random path. At each step, pick a random number of pixels to walk, followed by a random turn, anywhere in between 0 and 359 degrees. Allow the user to choose how many random steps the turtle will take. Adjust your program to allow the user to choose further parameters which direct the random walk. If this idea interests you, look up the term "random walk."