## ***3.3 Loop Control Elements***

*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:
```

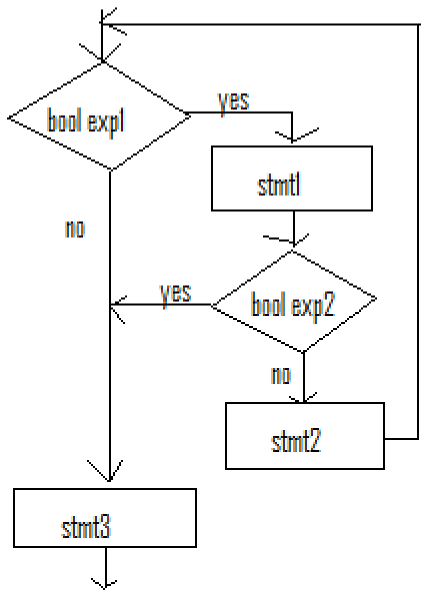
and use the break statement to get out of the loop whenever they want to. Similar to this:

```
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.

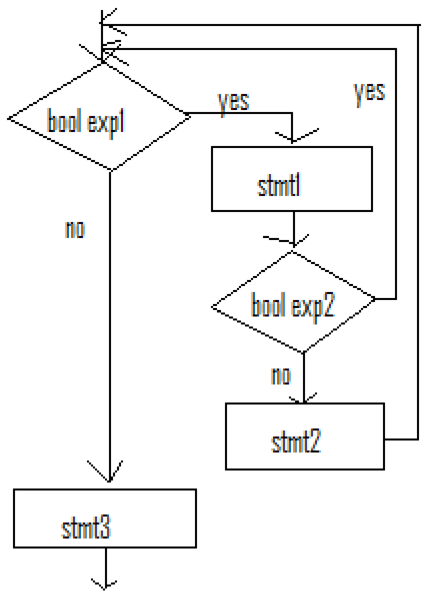
*Continue*

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.
```

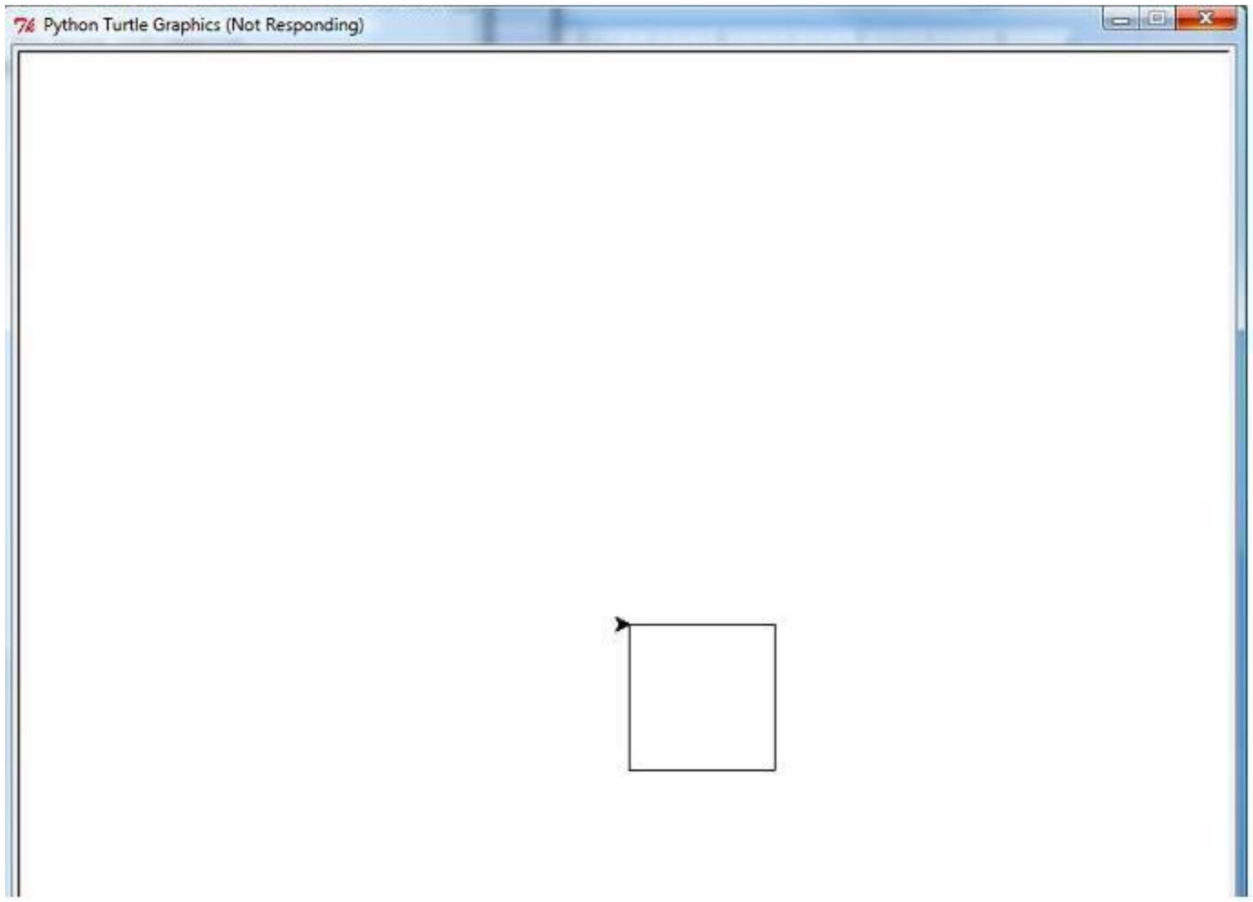
The easiest way to learn with the turtle is to try out examples. Let's look at four examples.

*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.

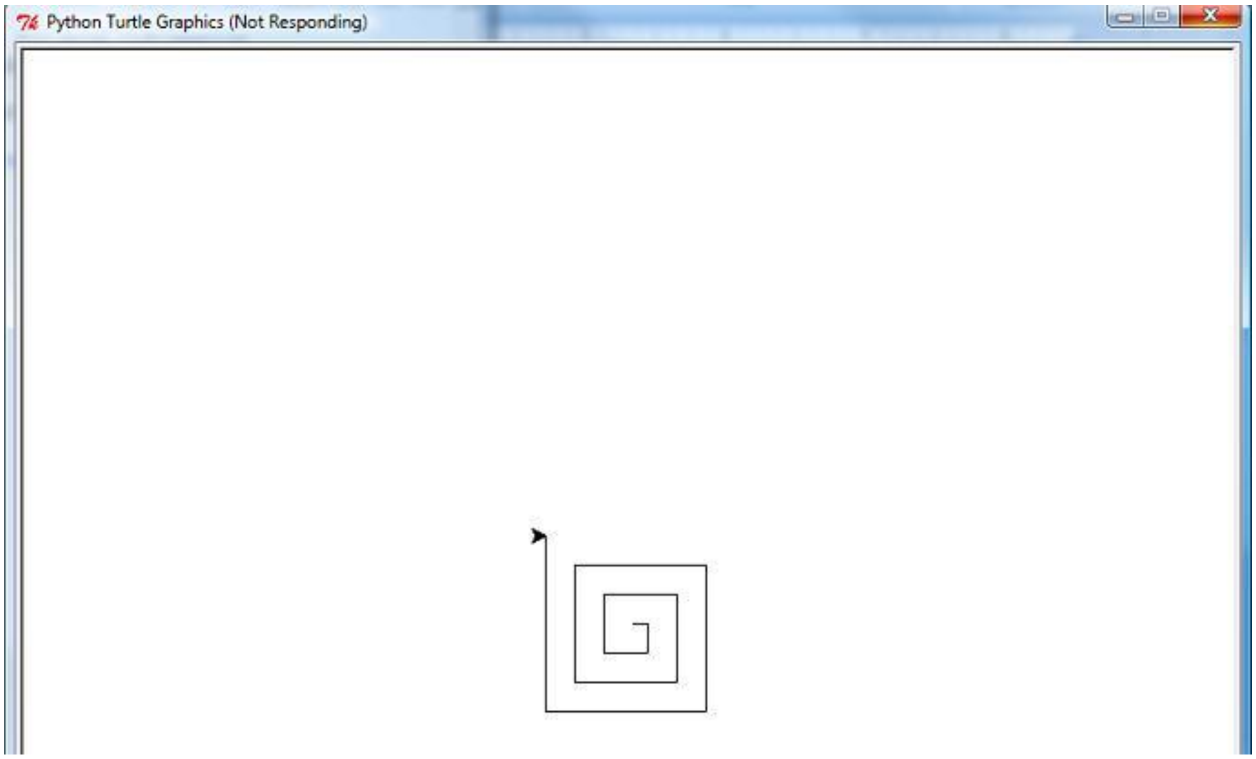
```
import turtle

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:

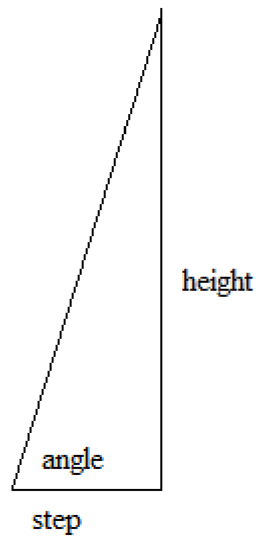
```
import turtle
import math

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:

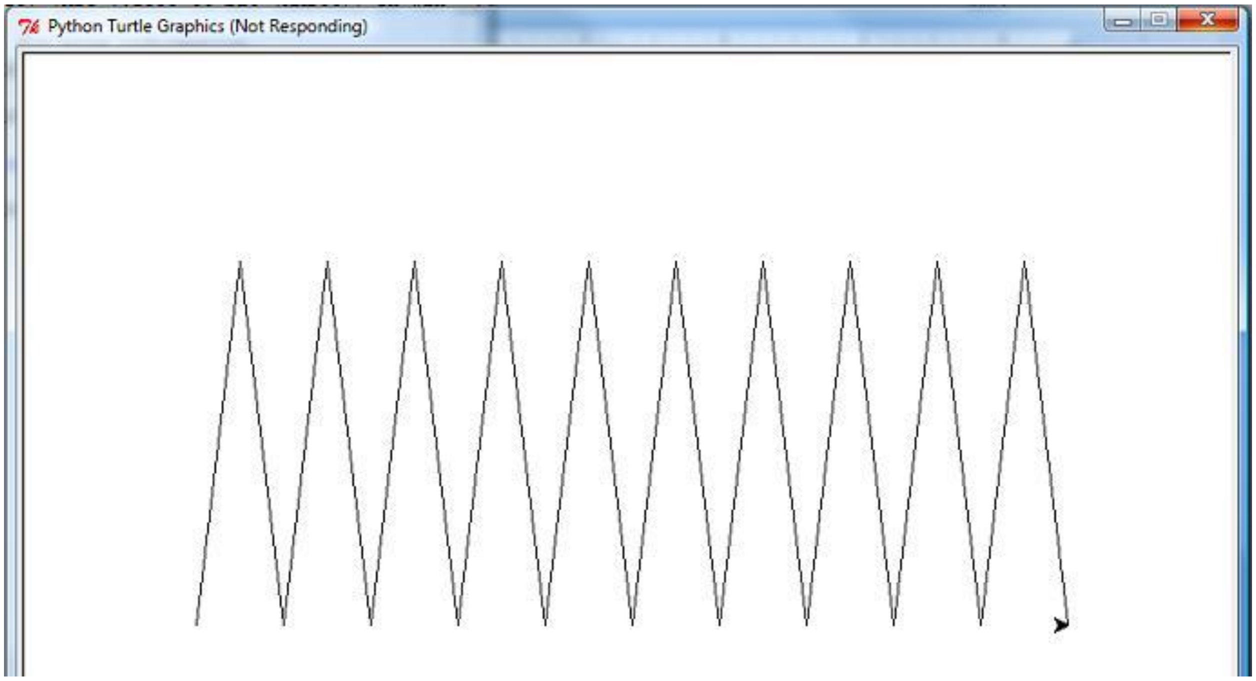


Using trigonmetric ratios, we find that $\tan(angle) = \frac{height}{step}$. Using the inverse tangent function, it follows that $angle = \text{atan}\left(\frac{height}{step}\right)$. The peak angle in the picture is the complement of angle. The Pythagorean Theorem can be used to find the length of the 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.

```
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."