CPADS Programming Activity V – Due 10/30

“Let It Snow!”

So far we have covered the basics of creating functions and implementing (fixed) iteration. The next important programming concept is decisions, i.e. *conditional* execution. The most common structure for decisions is the if-else construct. In Python the syntax is:

**if condition:**

*true* **statements**

**elif condition2:**

*true2* **statements**

**...**

**else:**

*false* **statements**

Just as with functions and iteration, the body of each branch of the decision logic is indicated by *indentation*, otherwise any valid Python code may be used within the branch.

Finally, we can combine iteration and decision structures to create *conditional* iteration (a while loop). For conditional iteration, the loop will execute *until* the condition statement becomes false. In Python, the syntax is:

**while condition:**

**statements to execute** *while* **condition is** *true*

**1. User Input**

Typically, one important aspect of a program is to obtain input from the user. This can be done either graphically via the mouse, or textually via the keyboard. In Python, the command to get user input – ***as a string*** – from the keyboard is

*var* **= input('***prompt****')***

where *var* is a variable used to store the user input and *prompt* is a literal string to be displayed to the user usually explaining what type of value you wish them to enter.

To convert the string into a number, Python has the following functions:

*n* **= int(***var***) –** (tries) to convert the string *var* to an integer that is stored in *n*

*n* **= float(***var***)** – (tries) to convert the string *var* to a float (i.e. decimal) that is stored in *n*

* Open **IDLE** and enter the following bold Python commands in the shell window noting the output:

>>> **var = input('Enter the number 123.4:')**

Enter the number 123.4: **123.4**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

>>> **int(var)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

>>> **float(var)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Note that if you try to convert a string of incompatible type, Python returns an error message called an *exception*. As with many other programming languages, Python provides a mechanism for the programmer to decide how to handle exceptions when they occur (via a try/except structure) but we will not deal with that in this course.

**2. Snowflake**

An interesting geometric figure known as a *Koch* curve can be drawn using a *recursive* function, i.e. a function that calls itself. One important consideration that must be observed when dealing with recursive functions is to ensure that they eventually reach a *termination* point, otherwise they will recurse infinitely eventually crashing the program. To determine where to terminate, a recursive function always has some type of decision logic within the function. The recursive algorithm for drawing a Koch curve of length *L* is show below. Note the different drawing done in each branch of the conditional statement.

**Koch(L):**

**if L > 2:**

Draw a **Koch** of length *L*/3

Turn left 60 degrees

Draw a **Koch** of length *L*/3

Turn right 120 degrees

Draw a **Koch** of length *L*/3

Turn left 60 degrees

Draw a **Koch** of length *L*/3

**else:**

Draw a **line** of length *L*

Using the following skeleton code as a starting point

**# Load TurtleWorld functions**

**from TurtleWorld import \***

**# TODO: Koch function**

**def Koch(t, length):**

**pd(t)**

**# TODO: Input function**

**# Main program function**

**def main():**

**# Create TurtleWorld objects**

**world = TurtleWorld()**

**turtle = Turtle()**

**turtle.delay = 0.001**

**# Snowflake size**

**size = 100**

**# Draw graphics**

**Koch(turtle,size)**

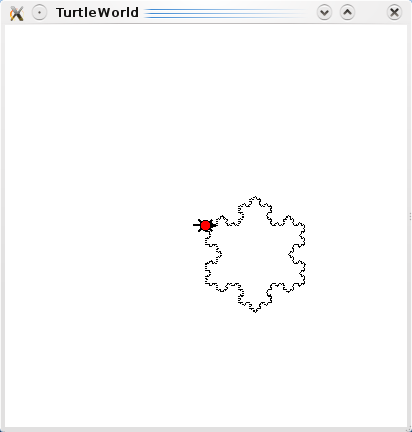
**# Press enter to exit**

**key = input(‘Press enter to exit’)**

**world.destroy()**

**# Call main function**

**main()**

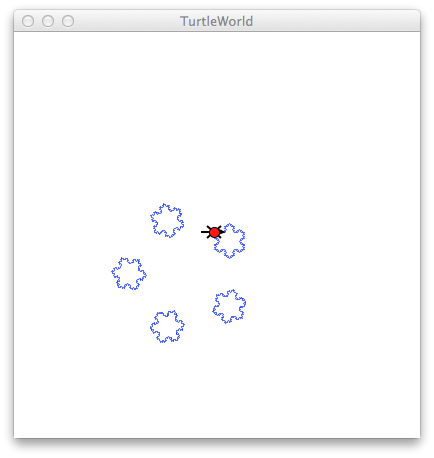
* Write a function named **Koch()** that takes two parameters – **t** for the drawing turtle and **length** for the length of the curve. Save the program as **snowflake.py.** Show the instructor your output.
* Extend the previous program to add a function called **get\_input()** that has one parameter – **prompt** that will contain the string to display to the user in the IDLE shell. The function **get\_input()** should *return* the **integer value** of the user input (assume the user will enter a string representing an integer). In the **main()** function, replace the assignment statement for **size** to set its value using a function call to **get\_input()**. Only draw the curve if the length entered is greater than zero (a programming technique known as *data validation*). Hint: The data validation should be done in **main()** not **get\_input().**
* Extend the previous program by adding code to the **main()** function to draw a *snowflake* using 3 Koch curves, see the sample output below. Hint: Consider each Koch curve as the side of a triangle and use a *loop*.
* Test your program using both valid values, e.g. 100, and invalid values, e.g. -2.

**3. Blizzard**

* Move the snowflake drawing code from the main program into a function called **draw\_snowflake()** that has two parameters – **t** for the turtle to use for drawing and **size** for the size of the snowflake to draw.
* Call the **get\_input()** function a second time in the main program to obtain the number of snowflakes the user would like to draw. Note: You will want supply a different prompt string as the *argument* in this call with appropriate text.
* Write a function called **draw\_blizzard()** that has three parameters – **t** for the turtle to use for drawing, **size** for the size of the snowflakes to draw, and **n** for the number of snowflakes to draw. The function should draw snowflakes at what would be the corners of a polygon with *n* sides.
* Add a function call in the main program to **draw\_blizzard()** (with appropriate arguments) that continually draws the snowflakes **AS LONG AS** the user has entered a positive number of snowflakes. (And not drawing anything if they subsequently enter a negative size). Hint: You can clear the screen with the command **world.clear()**
* Save the file as **blizzard.py**, print out and attach a copy of your program to this activity, and submit your source file through Marmoset (<https://cs.ycp.edu/marmoset>).
* A sample output run is shown below (note this is a ***single***execution of the program).

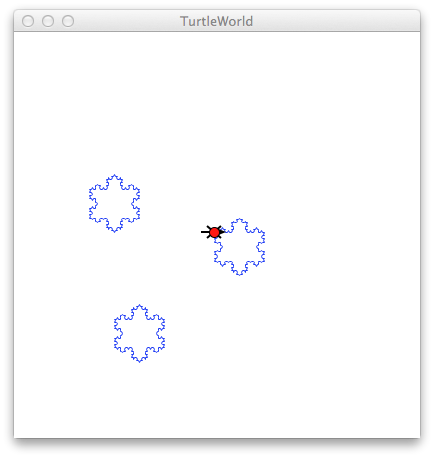
Please enter the number of snowflakes: **5**

Please enter the snowflake size: **30**



Please enter the number of snowflakes: **3**

Please enter the snowflake size: **50**



Please enter the number of snowflakes: **-2**

Thank you for playing in the snow.