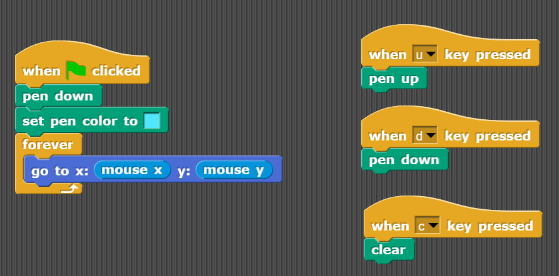
**Lab 1**

**Checkoffs:**

* **N/A**

**Blocks:**

**Kaleidoscope:**

****

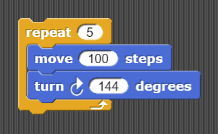
* Remember to set mouse x and mouse y to negative or not depending on which sprite you’re working with

**Lab 2:**

**Checkoffs:**

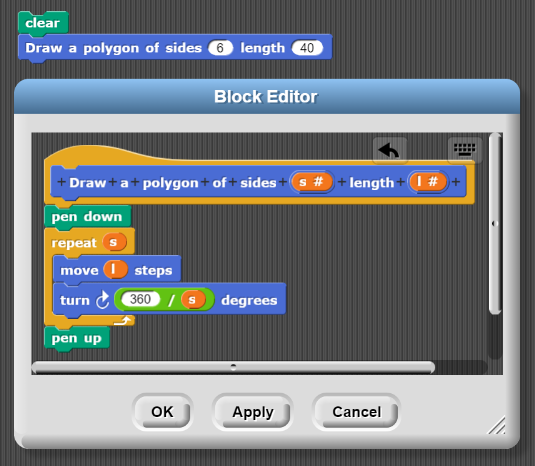
* ­Draw-square leaved flower
* Random walk
* Field of flowers

**Drawing a star: (Not checkoff)**



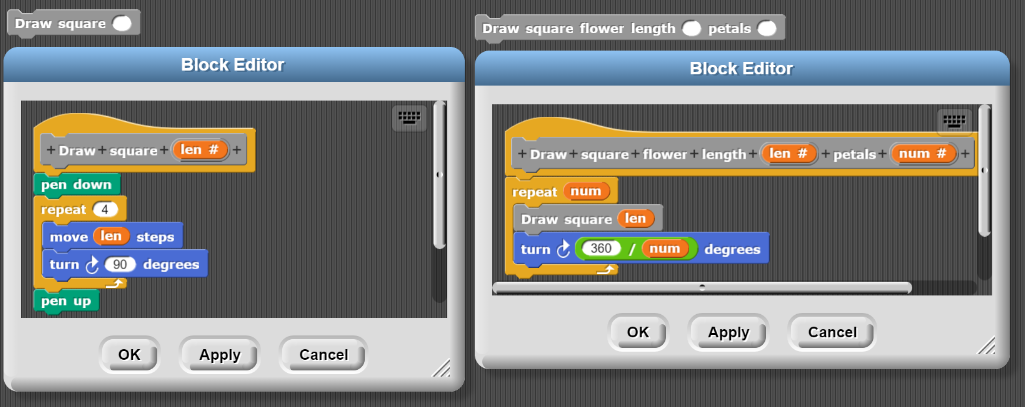
* Hint: 72 degrees is the turning angle required to draw a pentagon
* An isosceles triangle is drawn as an extension from the original pentagon and each ‘foot’ of the isosceles triangle is 72 degrees, meaning that the small angle of the triangle is 36 degrees.
  + Thus, the turning angle must be 144 in order to complete the star

**Draw polygon block: (Not checkoff)**

****

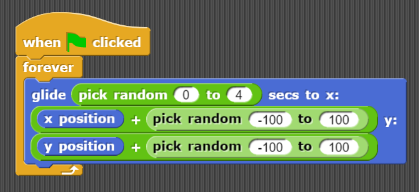
* Don’t forget about the “turn 360/sides degrees” bit

**Draw square-leaved flower:**

****

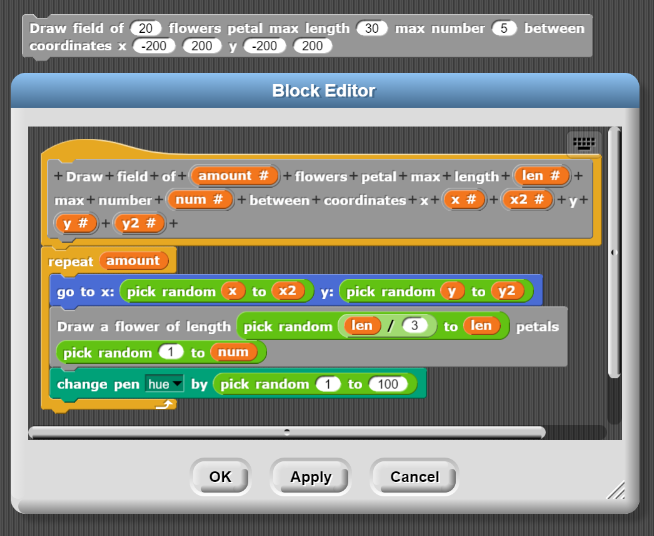
* Similarly, don’t forget to generalize the turning angle for a shape of any number of lengths

**Randomly moving character:**

****

* Yup, it’s random… like… ye

**Draw a field of flowers:**

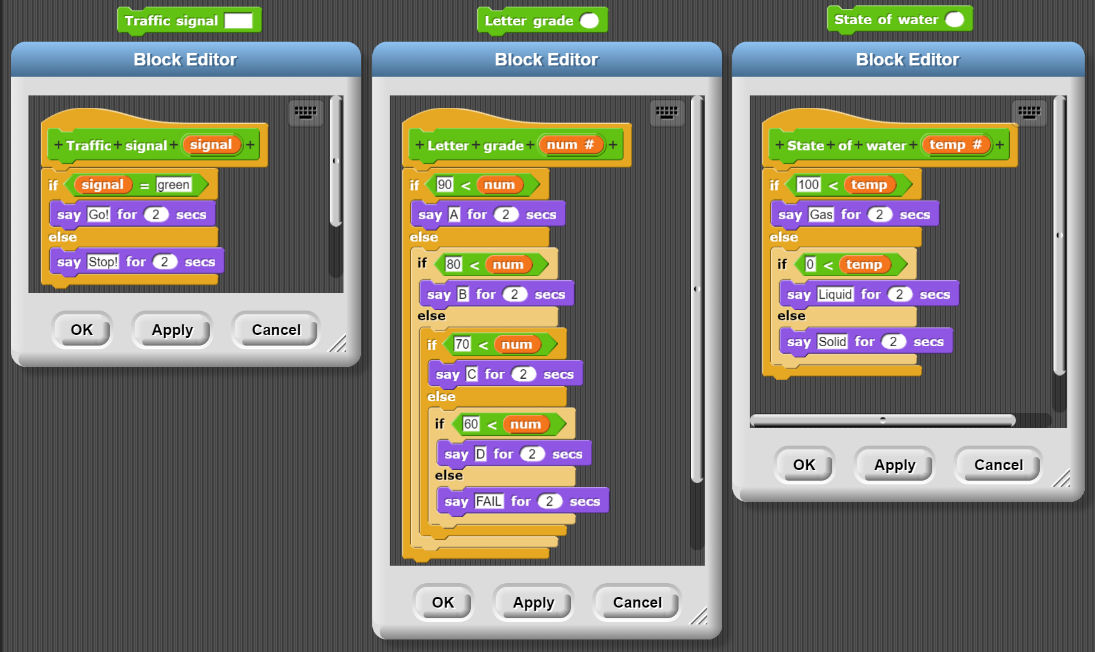
****

**Lab 3:**

**Checkoffs:**

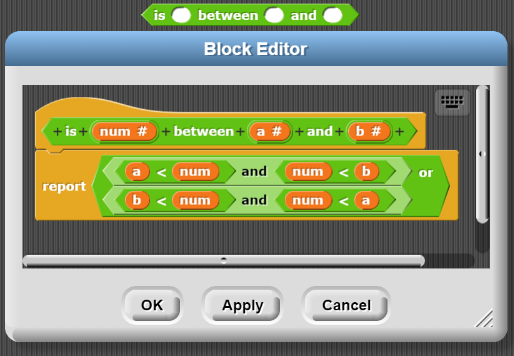
* Traffic, grade, water
* Is\_between\_and\_ block

**Traffic, grade, water:**

****

* Make sure that all the Booleans are correct
* **Note that** the grade one might have different actual grade boundaries

**Is\_between\_and\_:**



**Lab 4:**

**Checkoffs:**

* Valid-Date
* Sum-of-two smallest

**Valid date:**

* Honestly, I really can’t be bothered making this one -\_-

**Sum-of-two smallest:**



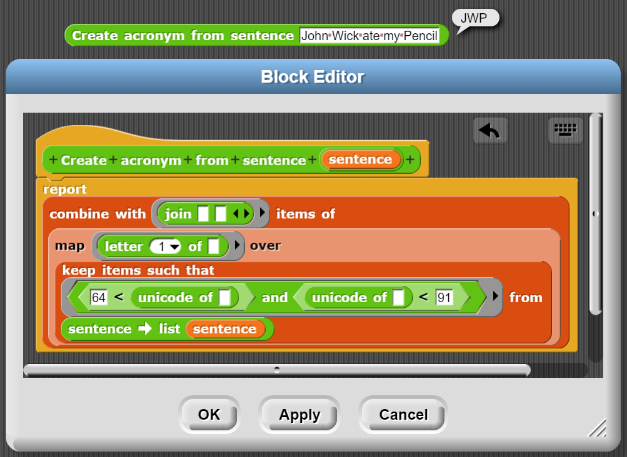
* Yup… I did it like a madman

**Lab 5**

**Checkoffs:**

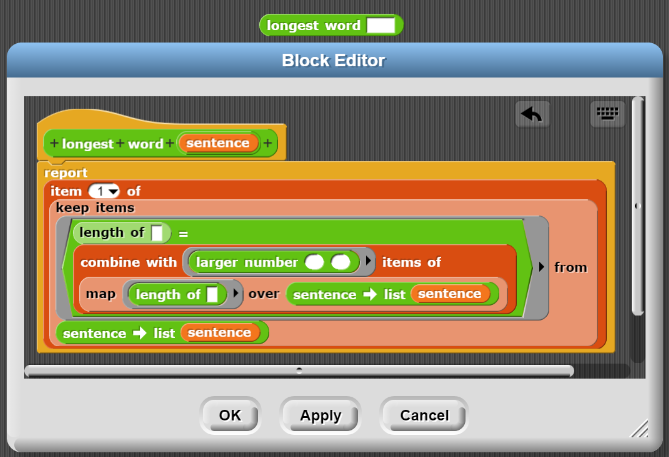
* Acronym
* Expand

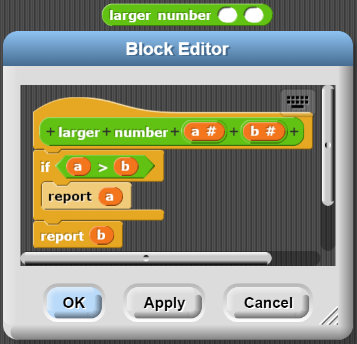
**Acronym:**

****

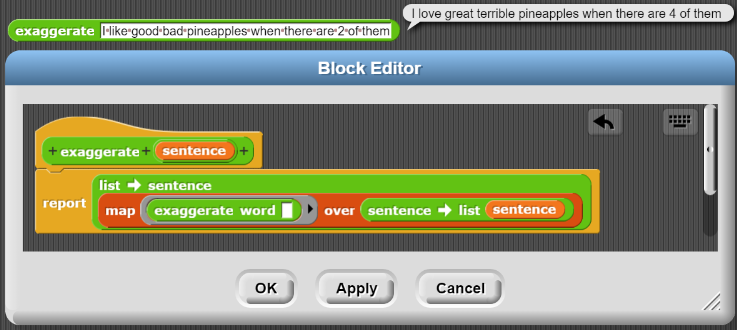
* **Steps as follows:**
  + Keep all letters with Unicode above that of capital “A” and lower than capital “Z”
    - Note that the Unicode of Z is 90 and A is 65
    - 
    - Also note that the Unicode of a word is the Unicode of the first letter of the word (apparently)
  + Map the [letter 1 of \_] block over the list to retain only the first letter of each kept word
    - These words should only be the ones with capital letters
  + Combine the remaining list with the [join \_ \_] block in order to create an acronym string
* Note that this task requires playing around with the Unicode blocks so yeah…

**Longest Word: (Not checkoff)**

****

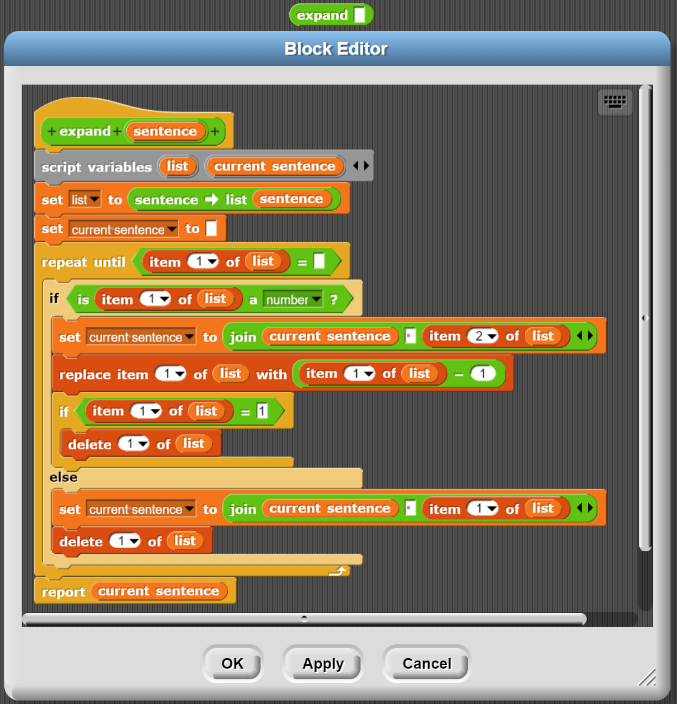
* **Steps as follows:**
  + Map [length of \_] block over the sentence that you convert into a list in order to acquire a list of numbers
  + Combine using custom [larger number] block the items of the list of numbers in order to acquire the largest length value
    - 
  + Keep only the words in the sentence-list with the same length as the longest length
  + Only take the first of the remaining sentence-list as you would like to report only one word
* Note that it is not necessary to report the first, you can report the last, I don’t think the question specified which one to report so it doesn’t HAVE TO BE the first longest length word.

**Exaggerate: (Not checkoff)**

****

* **Steps as follows:**
  + Map the [exaggerate word] block over the sentence (which has been converted into a list)
    - 
  + Convert everything back to a sentence

**Expand:**

****

* **Idea:**
  + Basically, I walk through the sentence as a list, and I add the words to a new sentence as I remove those words from the sentence
  + If I encounter a number then I add the word following the number and reduce the number by 1 without removing items from the sentence
* Create the variable “current sentence” as a method of keeping track of the sentence as you build it
* Create the variable “list” which is the sentence converted into a list to allow removing of items from the sentence
* Create [repeat until] loop that only concludes when the “list” variable becomes empty
  + If the first item is a number, append the second item to my “current sentence” variable and reduce the value of the first item by 1
    - If the value of the first item reaches 1, then the item gets deleted allowing the program to move on
  + If the first item is a word, then simply append it to the “current sentence” variable and delete it from the list

**Lab 6**

**Checkoff:**

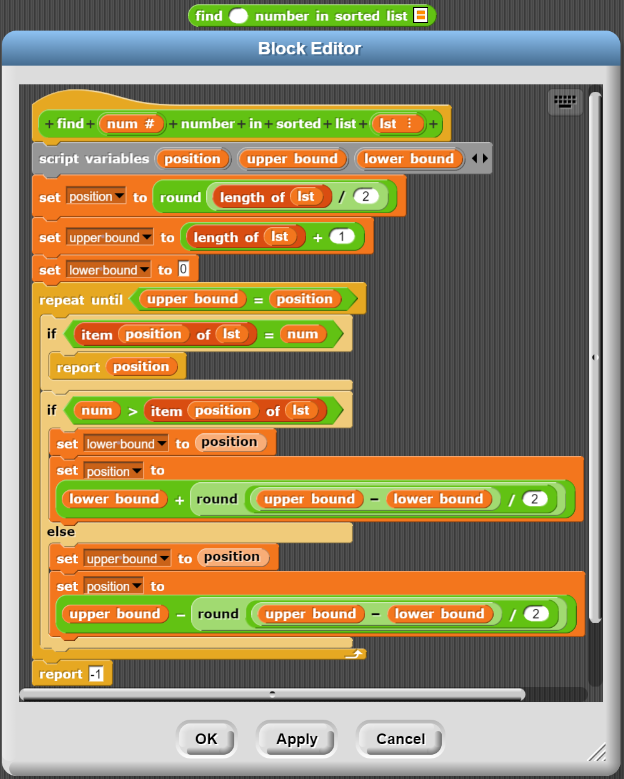
* Find the number X in unsorted/sorted list

**Find number in sorted/unsorted list:**

* **Unsorted list:**

****

* + Straightforward, just walk through the list in order and add numbers to the “tracker” variable as you go
  + If the number is a hit, then report the “tracker” number, if you reach the end without a hit, report “-1”
* **Sorted list:**



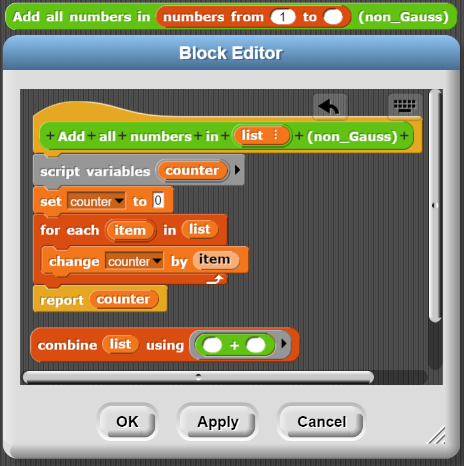
* + Following the idea behind the binary search algorithm
    - There is a “position” variable, set to half the length of the list
    - Then there are the “upper bound” and “lower bound” variables
      * Both set beyond the scope of the list in order for the program to check the first and last items of the list without terminating
  + **Steps:**
    - The “position” is checked, and if it’s a hit, we report the “position”
    - Otherwise, depending on whether the number at the position is higher or lower than the number we’re looking for we change either the “upper bound” or “lower bound” and then find the middle number between the upper and lower bounds and set that to the new position
    - Repeat until either the number is found or the position is equivalent to the upper bound indicating the nonexistence of said numbers

**Lab 7**

**Checkoff:**

* Gauss and Non-Gauss [add all numbers in] block

**Add all numbers in list (Non-Gauss):**



* The process is simple
* Create a “counter” variable that keeps track of the sum
* Add the numbers of every item of this long list to the “counter” and report the counter at the end
* There is a simpler (but also non-constant) solution of using the [combine] higher order function

**Add all numbers in list (Gauss):**



* Follow the provided formula of
* Note that these only works if the provided list is a list from 1 to N incrementing by 1 at a time

**Lab 8**

**Checkoff:**

* Test block on [merge column]
* 2048 game

**Not Applicable:**

This is one of the labs where the approach towards the question may differ from person to person and cannot be clearly illustrated by a clear example answer

**Lab 9**

**Checkoff:**

* Next move for computer block
* Show working game

**Not Applicable:**

This is one of the labs where the approach towards the question may differ from person to person and cannot be clearly illustrated by a clear example answer

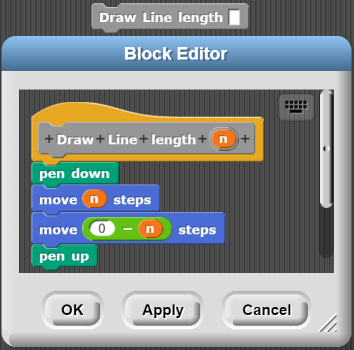
**Lab 10**

**Checkoff:**

* Random(crazy) tree
* Snowflake

**Abstractions:**

* **Draw Line length \_:**

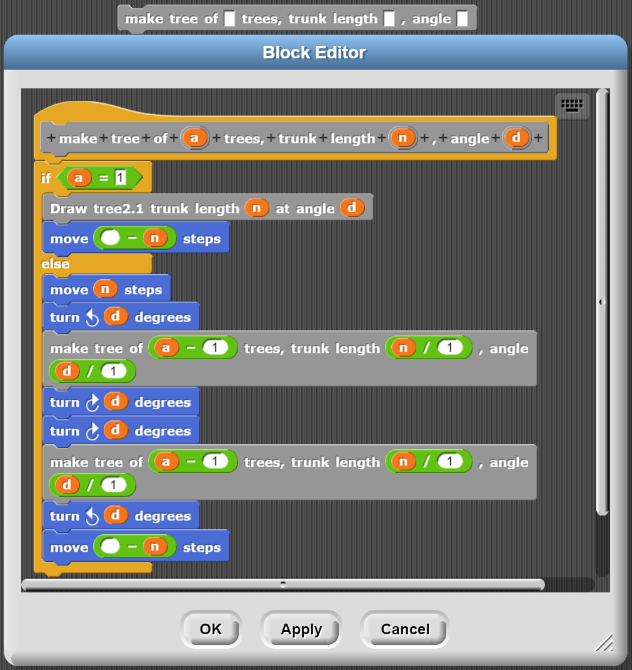


* + Draws a simple line of given length and returns to original position
* **Draw Tree type 2.1**



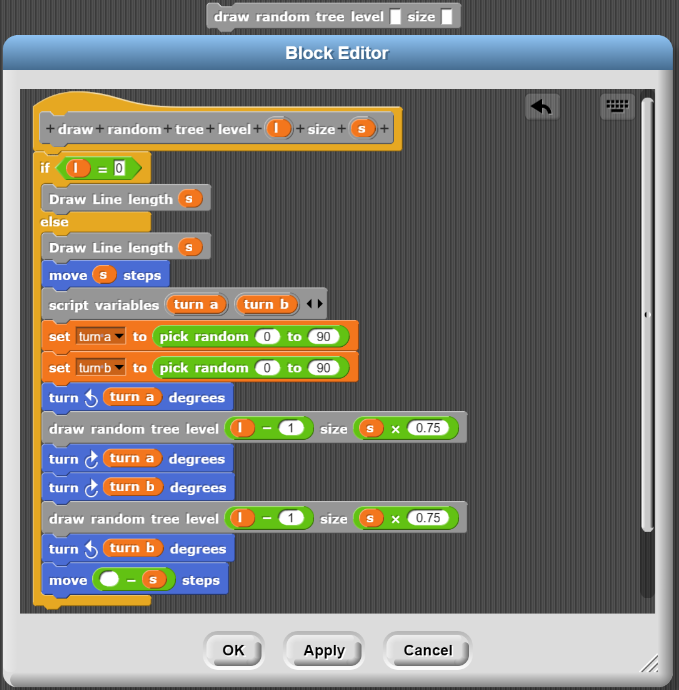
* + Draws a simple tree of given length (of main trunk) and with its branches at a given angle
  + The sprite finishes at the end of the main trunk and the base of the branches
  + **Note that** the implication of this simple tree could vary from person to person, and therefore the implications of the following questions could be different

**Recursive tree: (Not Checkoff)**



* This is a simple recursion where the base case consists of drawing a simple tree
* The recursive case involves the movement that will construct the “trunk” of this recursive level, and then construct a Recursive tree at the tip after a rotation of the given angle
* It then returns to the starting point having drawn two branches of the Recursive tree at a lower level

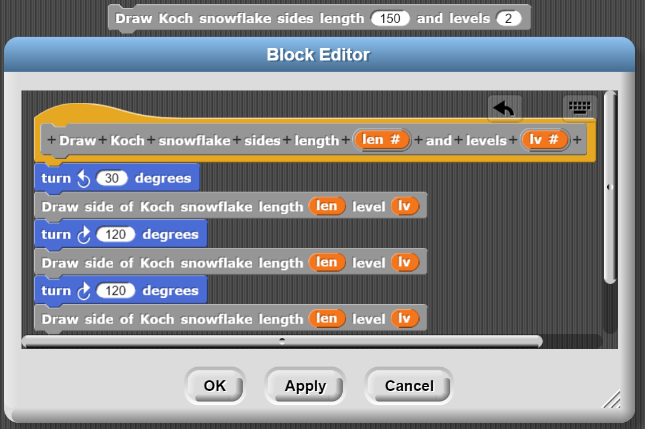
**Random(crazy) Tree:**



* + The base case is simply drawing a branch and occurs at level 0
  + The recursive case involves drawing the initial trunk and then remaining at the tip of the trunk before recursively drawing more random trees as the branches
    - **Note that** the first and second turns are stored as randomly chosen variables “turn a” and “turn b”
    - This allows you to return to the neutral position after your randomized turn
  + Then the sprite returns to the original position as a final move

**Snowflake:**



* + The base case is simply drawing a line and remaining at the end point
  + The recursive case involves drawing 4 more sides of the Kock snowflake to make up one length
    - Each at 1/3 of the original side length
    - Each at 1 lower level
  + The turning angle should be 60 degrees because such is the turning angle of an equilateral triangle
* **The Full snowflake:**
* ****

**C-Curve: (Self-Check Checkoff)**



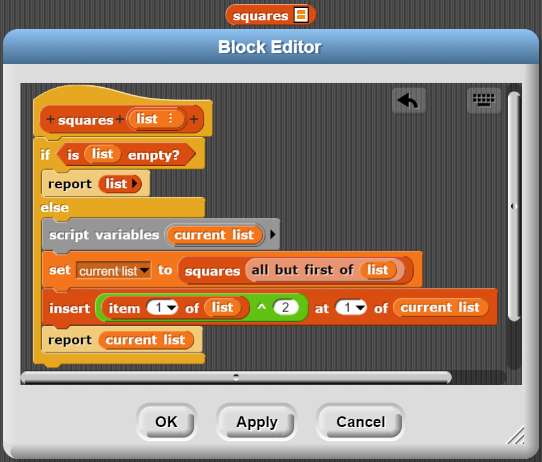
* + The base case is simply creating a line and facing in the direction the line is created in
  + The recursive case you must notice that the initial turning angle is different every level, therefore there must be a turn included as the first step of the recursive case
    - Note that the turning angle is observed to be 45 degrees
    - Also note that the length of the recursive call is found by multiplying the length by or doing as per Pythagoras where and are the same length and is the “len” variable
  + You notice that there are two recursive calls, one after the initial turn, another in 90 degrees and construct another recursive case
  + Then a final 45 degree turn to revert the sprite into it’s initial orientation

**Lab 11**

**Checkoff:**

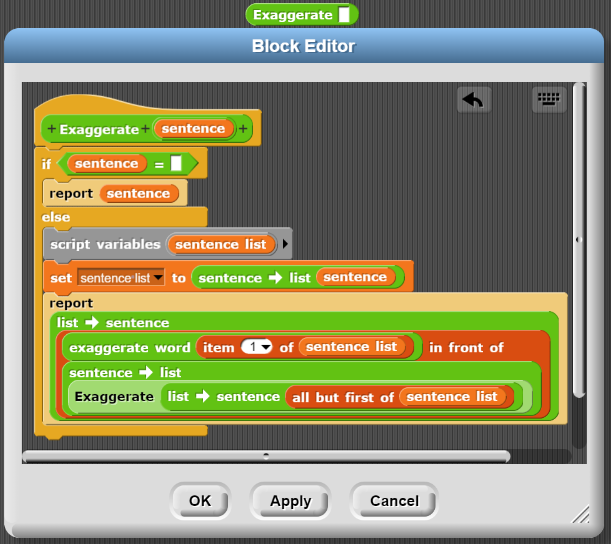
* The recursive [numbers] block
* The recursive [ends-e] block

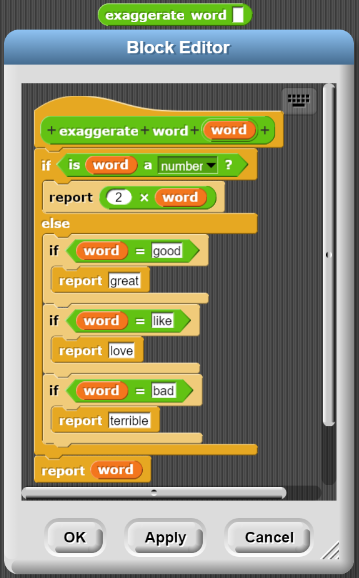
**Squares: (Not Checkoff)**



* + **Base case:**
    - Simply report the empty list should the list be empty
  + **Recursive case:**
    - Create a new variable and set the new variable to be the recursive call on the reduced “list”
      * All but first of the current “list”
    - Square the first item of the “list” variable and append that to the front of the “current list”
    - Report the “current list” which should have the first item of the list squared, and the rest of its items recursively squared by further calls

**Exaggerate: (Not Checkoff)**



* + **Base case:**
    - If the sentence is empty, return the empty sentence
  + **Recursive case:**
    - Basically, you take the [exaggerate word] block that we have constructed in a previous lab, and apply it to the first word of the sentence
      * 
    - The rest of the sentence list becomes exaggerated via a recursive call on the rest of the words of the sentence
      * The sentence is shortened by converting it into a list, removing the first item of the list, and then converting it back to a sentence
  + **Note that** there is a much less convoluted way of doing this by working directly with sentences (without the list-sentence conversion), however, that involves using imported blocks from outside of “import tools”

**Ends-e:**



* **Base case:**
  + If the list is empty, just return an empty list
* **Recursive case 1:**
  + If the first item of the list ends with the letter e, then report the word appended in from of the recursive call on Ends-e on the rest of the list
* **Recursive case 2:**
  + If the first item of the list doesn’t end with the letter e, then report the recursive call on the rest of the list

**Numbers:**



* + Pretty much the same as “ends-e”
  + Except the condition is that if it’s a number

**Subset of list: (Not checkoff)**



* + **Base case:**
    - If the list is empty, then there is only one empty set within the list
  + **Recursive case:**
    - This is the tricky bit; you must recognise that each subset which contains a specific element (say for example “apple”) will be the exact same as the number of subsets that do not contain that element
      * Also, note the existence of an [append] block which joins two lists together, this block must be imported to avoid a really long and convoluted solution
      * Therefore, there should be 2 components that join together when you report it
    - The first component is the recursive case in which it’s a call upon [subsets of list] with and item removed from the “list”
      * **Note that** I have chosen for the removed item to be the first for ease of usage, but it should be fine to have the last item removed or some other variation
    - There is the other case in which it’s the exact same subset as the one formed by calling [subsets of list] upon all but first of the “list”, except with the removed first item appended to the first of each subset, this forms the second half of this recursive case
      * **Written as** [map ([[item (1) of (list)] in front of \_] over ([subsets of list ([all but first of (list)])]

**Lab 12**

**Checkoffs:**

* Ancestors
* Me and my descendants of J

**Note:**

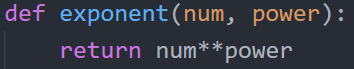
Need the starter file lol, maybe ask somebody afterwards?

**Lab 13:**

**Checkoff:**

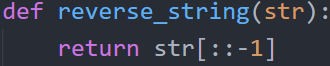
* Reverse\_string(str) function
* Palindrome(str) function
* C-curve(lv) function

**Exponent: (Not Checkoff)**



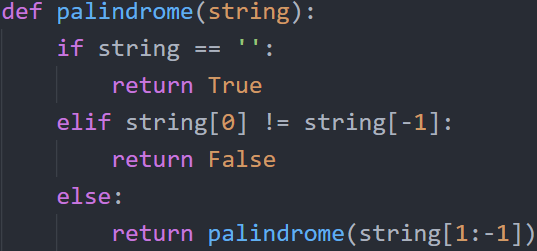
* + This is fairly straightforward; you simply use the [\*\*] operator function and return the value of it applied to the “num” and “power” variables

**Reverse\_string:**



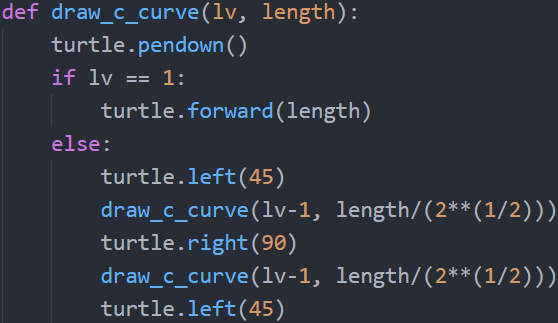
* + Using the power of list/string iteration (as strings are iterable) you can simply iterate through it in reverse order

**Palindrome:**

****

* Similar to our previously written [palindrome] block in snap, this one has 3 cases:
  + **Base case 1:**
    - If the string is now empty, then you have done your job checking every letter and the word is a palindrome, thus returning “True”
  + **Base case 2:**
    - If the first letter of the word does not equal the last letter of the word then the word is not a palindrome and you should return “False”
  + **Recursive case:**
    - If neither of the cases above have occurred, then you continue the check the remaining items of the string
      * “[1: -1]” suggest you begin from the 2nd item of the string (“1”) and the “-1” suggests to finish with the last item of the list non inclusive

**Draw\_c\_curve:**



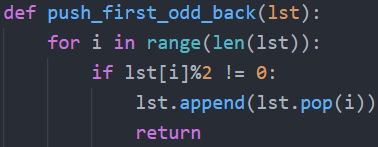
* Similarly written to our snap version of the [draw\_c\_curve] block, the first thing this function does is put the pen down
* **Base case:**
  + If the “lv” variable of the [draw\_c\_curve] function is 1, then simply move forward, drawing a simple line
* **Recursive case**
  + Recall that the rotation angle differs for every level, this suggests that the first thing the turtle sprite should do is rotate
    - Then angle of rotation may be observed to be 45 degrees
  + Then the turtle constructs a recursive case of 1 lv lower (hence the recursive call)
  + The turtle then rotates 90 degrees which may also be observed throughout the different cases
  + Another recursive case of 1 lv lower is constructed via a recursive call
  + The turtle then returns to its initial orientation via the [turtle.left(45)] call

**Lab 14:**

**Checkoff:**

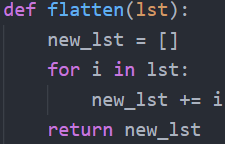
* Result of autograder

**Push\_first\_odd\_back:**



* + Fairly straightforward, there is a “for loop” that iterates through the numbers from 0 up to (but not including) the length of the list
    - These numbers represent the index of the number values within the list
  + If the item at that specified index of the list is an odd number, then it’s modulus would not be 0
    - Therefore, we pop the item at that index, and append it to the end of the list
      * Remember, the [pop] command function removes and item at a specific index, and also returns it
  + The return statement at the end is designed to simply exist the loop after the first item has been pushed to the back

**Flatten:**



* + The goal of flatten is to simply turn a list of lists into a single list
  + So, I simply added the lists together into a “new\_lst” variable and returned this new list

Square\_of\_evens:

****

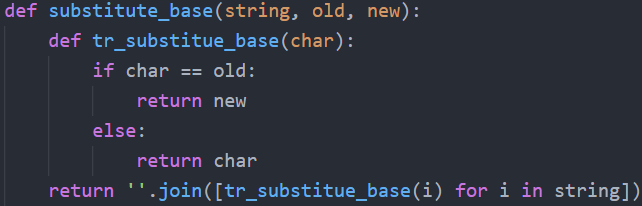
* + Using list comprehension, I simply squared [i\*\*2] every item within the list, given they pass the [if] condition that their modulus of 2 was equal to 0

**Nth\_power\_of\_evens:**

****

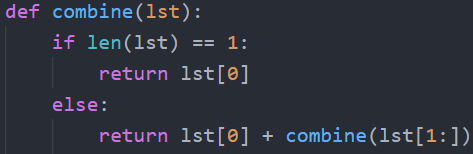
* + Similar to square\_of\_evens, I simply raised everything that fulfilled the condition of being even to the power of the provided “n”

**Substitute\_base:**



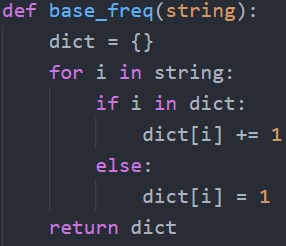
* + The helper function does its job quite simply:
    - If the provided character is the “old” character, simply return the “new” character, otherwise, return the original character.
    - Note that due to scoping, [tr\_substitue\_base] which is defined within [substitute\_base] is able to access the variables “old” and “new”
  + By applying [tr\_substitute\_base] to the string via list comprehension, I have acquired a list of all the letters of the old string, converted according to the given parameters
  + And then, using the [join] function, I am able to return a string with all of the elements of the list as its components.

**Combine:**



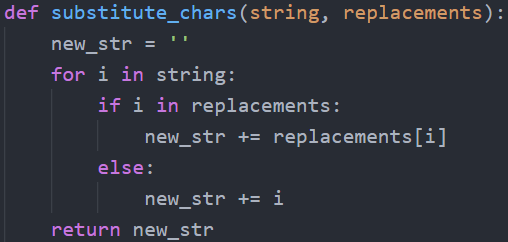
* + Combining all items of the list using the [+] function is fairly straightforward, there are 2 main ways to do it: with a loop or recursion, both of which are fairly elegant
    - **Base case:**
      * If the list only contains 1 item, then simply return that item
      * **Note that** it cannot be “if lst == []” because there will be an “unsupported operand” error when trying to add numbers or strings to a list
    - **Recursive case:**
      * Use the [+] operator function on the first item of the list and the recursive call upon the rest of the list to return a result
  + If using iteration, then creating a new variable and [+] to it should work!

**Base\_freq:**

****

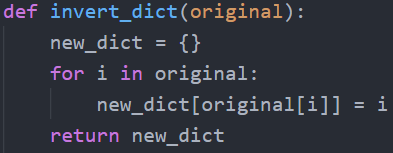
* + Iterate through the string and alter the created “dict” variable as characters appear
    - **Case 1:**
      * If the character has already been added to the dictionary, then add 1 to the count of that item
    - **Case 2:**
      * If the character has not appeared yet, then create a key within the dictionary of that item with 1 set as the count
  + Simply return the dictionary after completing the iteration

**Substitute chars:**

****

* + Create a “new\_str” variable set as an empty string
  + Iterate through the items of the string to be replaced and:
    - If the character is within dictionary “replacements” it is then meant to be replaced and therefore should be replaced with the item within its key in replacements
    - If the character is not within the dictionary “replacements”, then we simply add it to the “new\_str” variable
  + Return the now-completed “new\_str” variable

**Invert\_dict:**



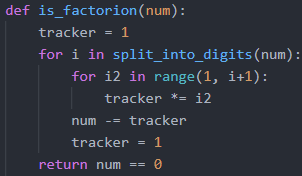
* + This question may appear challenging at first, but then you realize that you can simply iterate through the keys of the dictionary
    - Create a new dictionary
    - Simply iterate through the keys of the “original” dictionary
    - Add the values of the “original” dictionary as keys and then setting the key to become the value of the “new\_dict”
    - Return the “new\_dict”

**Lab 15**

**Checkoff:**

* Autograder results

**Is\_factorion:**



* + A factorion is a number of which the sum of the factorials of the individual digits is equivalent to the number itself
  + **Abstract**
    - Basically, what I am going to do is subtract the factorials of each of the number’s digits from the number itself, and then check to see if the number becomes 0 in the end
  + First, I create a “tracker” variable, this variable I use in my factorial calculations
    - Note that the “tracker” variable is set as 1, because [anything]
  + The first [for \_ in \_] loop, I input the digits of the number found using the [split\_into\_digits( )] function.
  + The 2nd [for \_ in \_] loop is used to find the factorial of the current “i” number, saved in the “tracker” variable
    - Note that it’s [range(start, **end+1**)] because the range function is not inclusive of the ending element
  + After finding the factorial of the current “i” number, the number is subtracted from “num” and the “tracker” variable is reset to 1, ready to find the factorial of the next digit
  + At the end, the function checks whether the number value is 0 and returns accordingly
    - If it is not 0, that means the number is not a factorion and [num == 0] should return “False”
    - If it is 0, that means the number is a factorion and [num == 0] should return “True”

**List\_all\_factorions\_between:**



* + This is a fairly straightforward function utilizing list comprehension that simply applies the [is\_factorion] function over the values provided by the [range] function

**Is\_pandigital:**



* A pandigital number is a number of which every number up to the length of the number is contained within the digits of said number
  + There cannot be repeated digits within this number
  + The highest number within this pandigital number has to be
* **Abstract:**

**Lab 16**

**Checkoff:**

* Izzle function
* Top\_n\_words function
* Print\_top\_n\_words function
* Apply\_language\_game function

**Izzle:**

