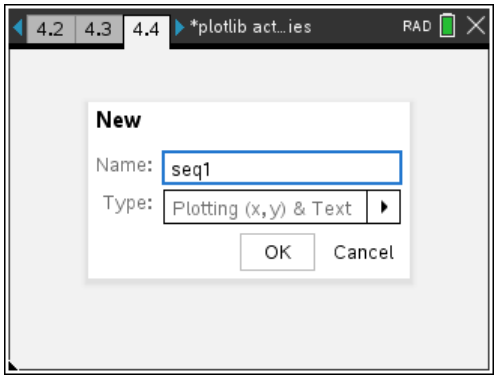
TI PlotLib **Sequences**

This activity demonstrates plotting mathematical sequences using the **TI PlotLib** module in two different ways: by plotting individual points and by plotting lists.

Part 1 - Start with an arith**metic** and a geo**metric** sequence in a single program. Begin a Python program using the '**Plotting (x,y) & Text**' template from the **'Type:'** dropdown list. Our program is called **seq1**.



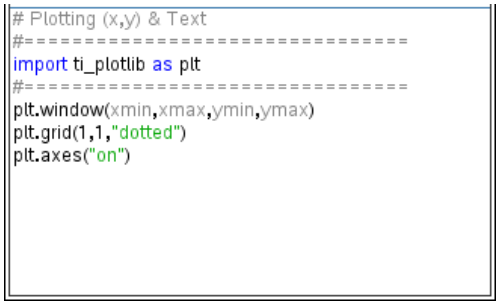
1. As explained in the Getting Started activity and worth repeating here, this template provides the unique **import** statement:

   import ti_plotlib as plt

   along with several 'Setup' functions.

   This type of **import** statement requires that all **ti_plotlib** functions be preceded by the alternate module name **plt.** This is called '*aliasing*' the module (give it a different, shorter name). When selecting **ti_plotlib** function from the menus, they will include this name at the beginning of the function as seen here in the first three functions provided.

   Note: these three statements, if used, must be listed in this order since each paints the canvas (screen) over the previous screen. First set the window, then draw the grid, then draw the axes on top of the grid.



```
# Plotting (x,y) & Text
#================================
import ti_plotlib as plt
#================================
plt.window(xmin,xmax,ymin,ymax)
plt.grid(1,1,"dotted")
plt.axes("on")
```

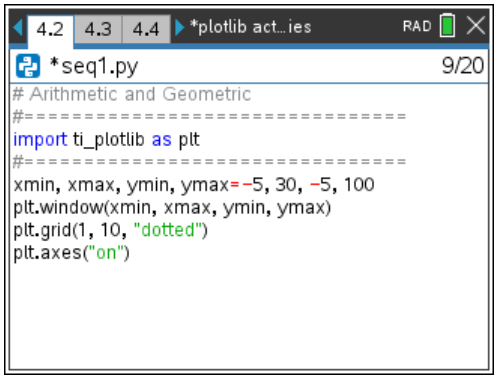
2. Use four 'window variables' to set the **plt.window** because you might need these values elsewhere in your code.

   xmin, xmax, ymin, ymax = -5, 30, -5, 100

   We're placing the origin in the lower left corner of the screen because we will be plotting points in the first quadrant.

   Having the four window variables assigned on a single line makes them easier to locate & edit and saves vertical space in your program.

   The **plt.grid((1, 1, "dotted")** values can be edited as needed. Change the y-scale value from 1 to to **10**. There are limitations on the window and these grid values *may* interfere with the run of your program.



```
4.2 4.3 4.4 *plotlib act...ies RAD
*seq1.py 9/20
# Arithmetic and Geometric
#================================
import ti_plotlib as plt
#================================
xmin, xmax, ymin, ymax=-5, 30, -5, 100
plt.window(xmin, xmax, ymin, ymax)
plt.grid(1, 10, "dotted")
plt.axes("on")
```

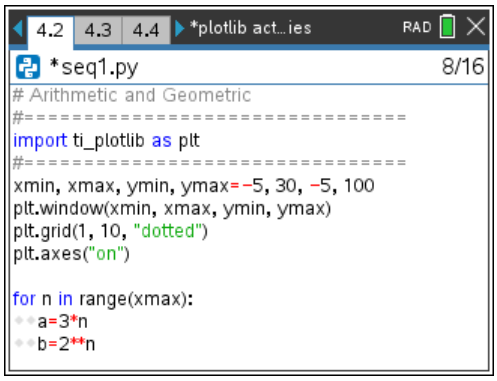
©2022 Texas Instruments Incorporated

3. Plot both an arith**metic** and a geo**metric** sequence in the same graph to compare them. Use a **for** loop that makes use of the window variable:

   for n in range(xmax):

   Note: change the loop variable from **i** *to* **n**.
   In the loop body, first assign two variables the value of an arith**metic**/linear term like **3n** and a geometric/exponential term like $\mathbf{2^n}$

   ```
   a = 3*n      # arithmetic
   b = 2**n     # geometric
   ```

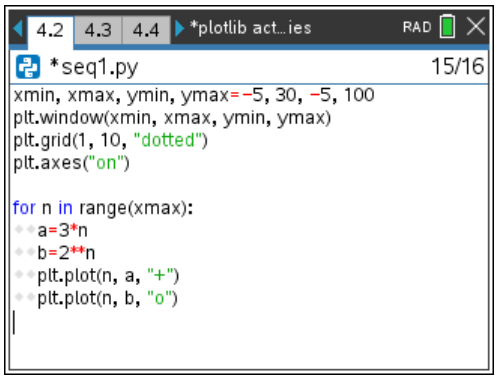


4. Plot each term using a different plot style ("mark"):

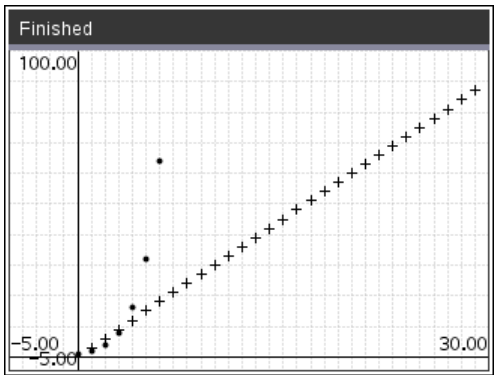
   plt.plot(n, a, "+")
   plt.plot(n, b, "o")

   Get **plt.plot(**x, y, "mark"**)** from **[menu] TI PlotLib > Draw**

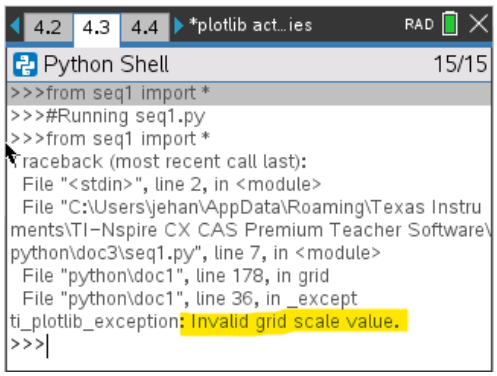
   Note: There are two ***plt.plot( )*** *functions on the menu. One is for plotting lists and the other is for plotting points.*
   The plot "mark" can be changed (by hand) and is limited to ".", "+", "x", or "o".



5. Run the program and you should see the two plots. One is the graph of points on a line and the other quickly disappears off the top of the screen. Which is which?



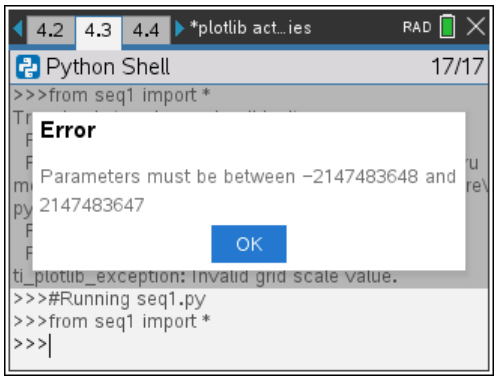
6. Try changing the window settings to see different views of these plots. Depending on your window values you may encounter the error seen here: We changed **xmax** to 100 and got the 'invalid grid scale' error because the grid lines would be too close together.
   Either:
   - change the **plt.grid**( ) *x-scale* value to a larger number (like 10) or
   - **#comment** the **plt.grid( )** function to hide the grid.



7. But even using **plt.grid(10, 10, ”dotted”)** still produces another error! Plotting limits values to be between -2,147,483,648 and 2,147,483,647 (that’s **2******31 - 1**).

   This is a special constraint built into the **ti_plotlib** module.

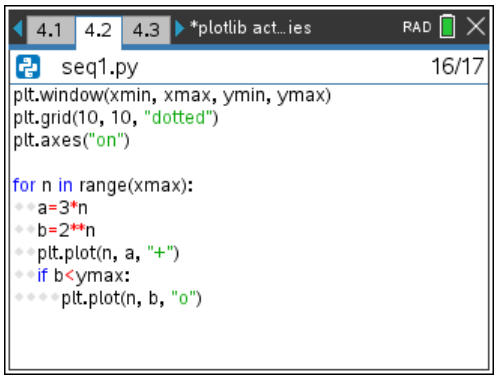
   The geometric/exponential sequence **b = 2******n** gets too large to plot. *Can you fix it? See the next step…*



8. We can fix this ‘*overflow*’ plotting issue by adding a ***condition***:

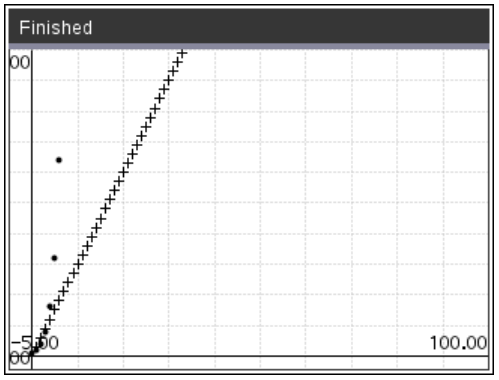
   if b < ymax:
   plt.plot(n, b, ”o”)

   since there’s no need to plot a point that’s not on the screen. You could also place the same restriction on the arith**metic** sequence plot statement.



9. Running the program now should produce a graph like the one shown here. Notice how the geometric sequence grows so quickly compared to the arith**metic** sequence. Both sequences go off the screen, but the arith**metic** sequence does not exceed the **ti_plotlib** upper limit.

   In the next section you will use most of this program to plot a different sequence, so we will make a copy of it…



10. **Part 2: The Collatz Conjecture**

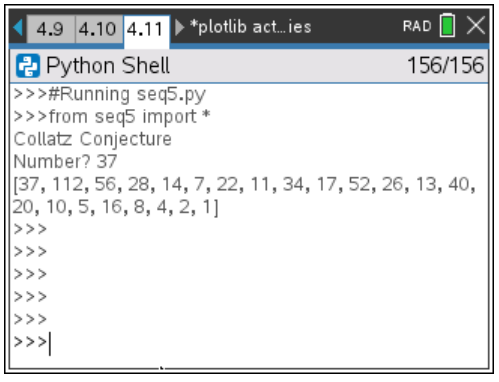
    The final sequence in this activity is *based on* the Collatz Conjecture which is also discussed in the **TI Codes Python** materials.

    The Collatz Conjecture goes like this:

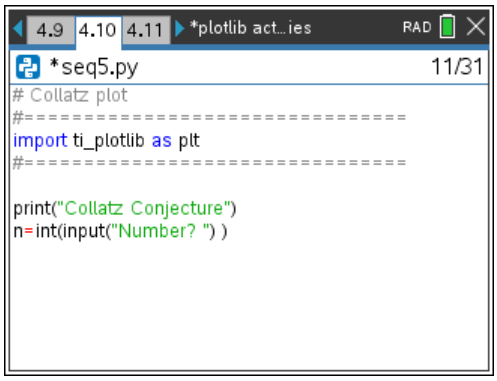
    0. Start with any counting number (we will use **input( )**).
    1. If it is even, then divide it by 2
    2. Otherwise, multiply it by 3 and add 1
    Repeat steps 2 and 3 with the new number.

    The Conjecture (guess) is that all numbers eventually will become 1 but *this has not been proven… yet*.

    Starting with the number **37**, we get the sequence shown to the right in the Python Shell. Let’s **plot** some Collatz sequences.



11. This program will differ greatly from the previous one, but we start with the usual **plt.** setup statements so make a copy of your last program (Harmonic sequence). Ours is named **seq5**.
Press **[enter]** a few times after the **import** statement to make room for some new code: the plan is to make two lists and then use **plt.scatter( )** or **plt.plot( )** to plot the lists rather than one point at a time as was done in Part 1.

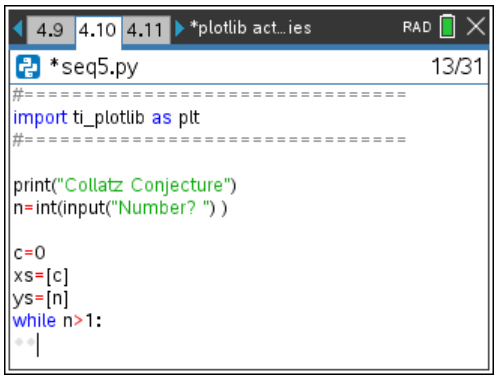


Write the first two statements shown:

```
print("Collatz Conjecture")
n = int(input("Number? ") )
```

12. Initialize a counter variable **c** to be **0**.

```
c = 0
```



Create two lists:

xs is the list of counters, starting with **0**.

ys is the list of terms in the Collatz sequence for n, starting with **n**.

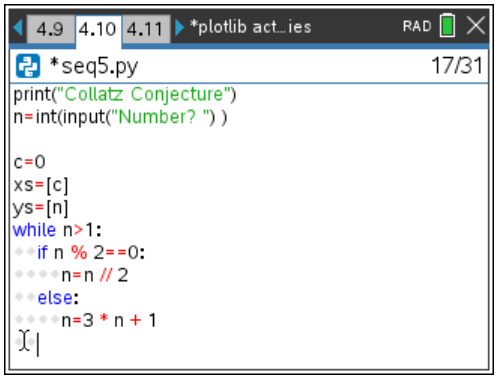
```
xs = [c]
ys = [n]
```

We're going to make a big assumption here: eventually the sequence will reach 1. After all, it has never *not* happened, right?

```
while n > 1:
```

13. Write the **if… else** structure that processes the Collatz algorithm:

```
while n>1:
  if n % 2 == 0:  #    % is 'mod'
    n = n // 2      # integer division
  else:
    n = 3 * n + 1
```

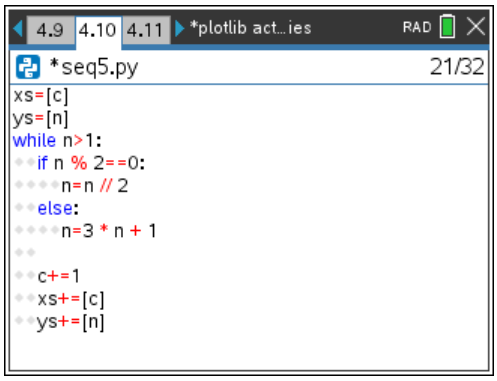


Note: use integer division (//) to ensure all values are integers and there are no rounding issues.

14. Still in the loop: increment the counter and add (**append**) the counter **c** and the number **n** to their associated lists. Note the use of 'addition' here to *append* a list to another list:

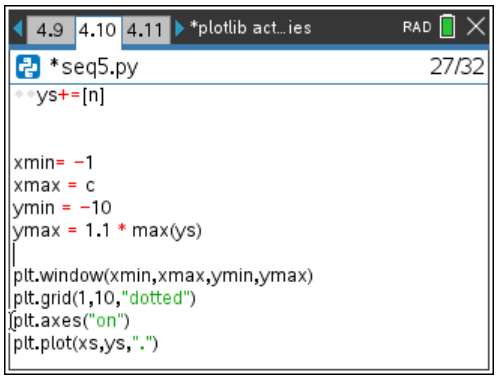
```
c += 1
xs += [c]   # same as xs.append(c)
ys += [n]
```

Note that these statements are part of the **while** *loop but not part of the* **else:** *block and are indented accordingly.*



15. When the **while** loop ends the sequence is complete so we're ready to plot the data.
Use a *special window* that depends on the data itself. We write each window variable assignment on a separate line for readability:

```
xmin =  -1
xmax = c     # the final counter value
ymin = -10
ymax = 1.1 * max(ys) # so all fit on screen
# 1.1* is 10 percent more than the largest
#  number in the list ys
plt.window(xmin,xmax,ymin,ymax)
```

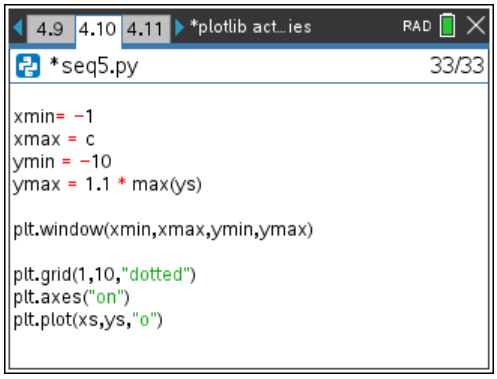


16. The last three statements control the grid and axes (they were here from the start) and then **plot** the lists using:

plt.plot(xs, ys, "o") will plot the lists with segments connecting the dots.

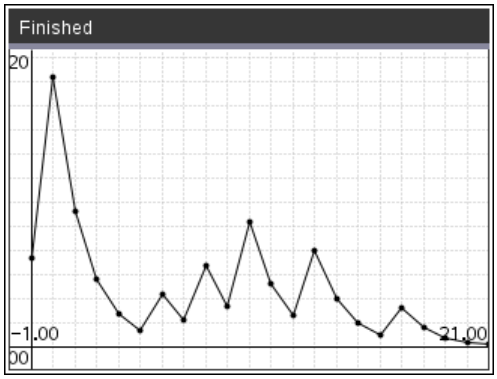
or use:

plt.scatter(xs, ys, "o") to plot just the points without the connections.



17. Run the program. At the **input** prompt enter a positive integer. The plot displays the unique progression of the Collatz sequence of numbers for your entered number and the **xmax** value on the screen is the number of steps it took to reach 1. Your starting value is the point on the y-axis (it was 37 for this image; **ymax** is 120, not 20).

Note: For some numbers (like 125) you might have a grid scale issue, so just comment the **.grid( )** *statement or find a scale that works.*



18. To display the initial value on the graph use:

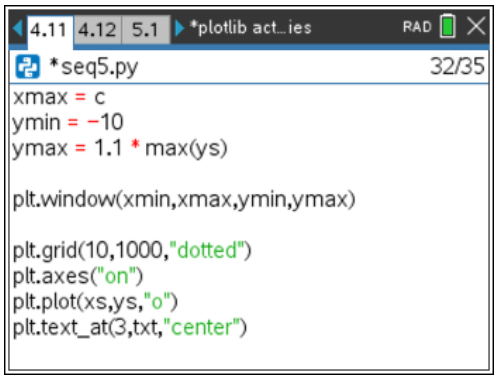
plt.text_at(row, "text", "align")

found on **[menu] TI PlotLib > Draw**

row is a number from 1 to 13 that you enter.

"text" is your text (or a string variable) to display

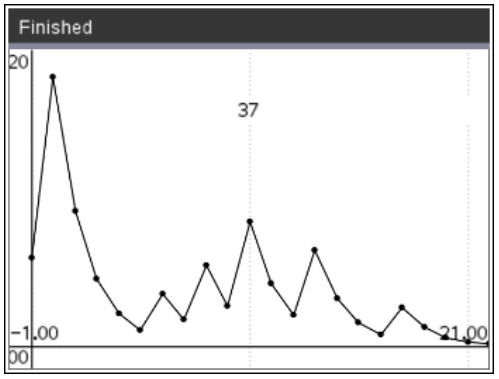
"align" can be "left", "center", or "right" selected from a pop-up menu.



To display the starting number for your sequence, add a statement right after the **input** statement that stores the value of **n** as a string:

txt = str(n)

str() is found on **[menu] Built-Ins > Type** and use that **txt** variable in the **plt.text_at( )** statement.



The '37' at the top center of this screen is the result of the **plt.text_at()** function:

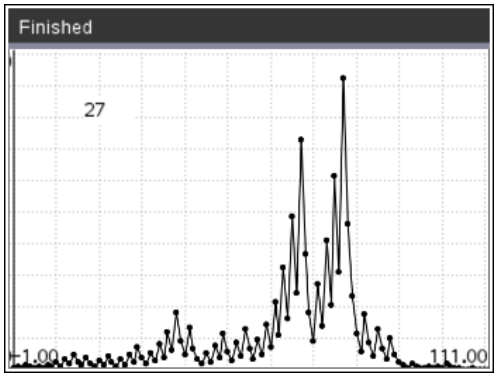
plt.text_at(3, txt, "center")

19. For more *precise* position of text you can use the alternate form:

plt.text_at(row, col, "text")

(which is not found on the menus).

Use a ***col****umn value from 1 to 30+.*



For this screen, **plt.text_at(3, 5, txt)** was used.

For other interesting integer sequences see https://oeis.org/, the **Online Encyclopedia of Integer Sequences**.