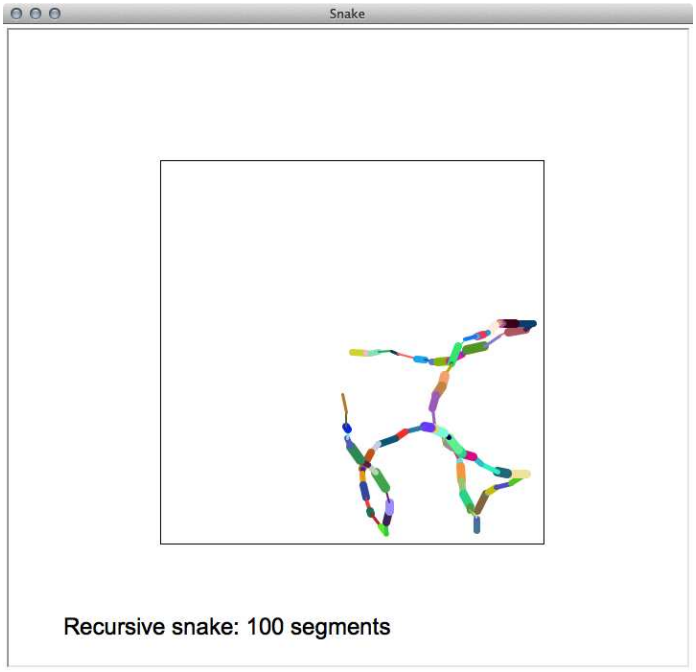
Computer Science I
Snake

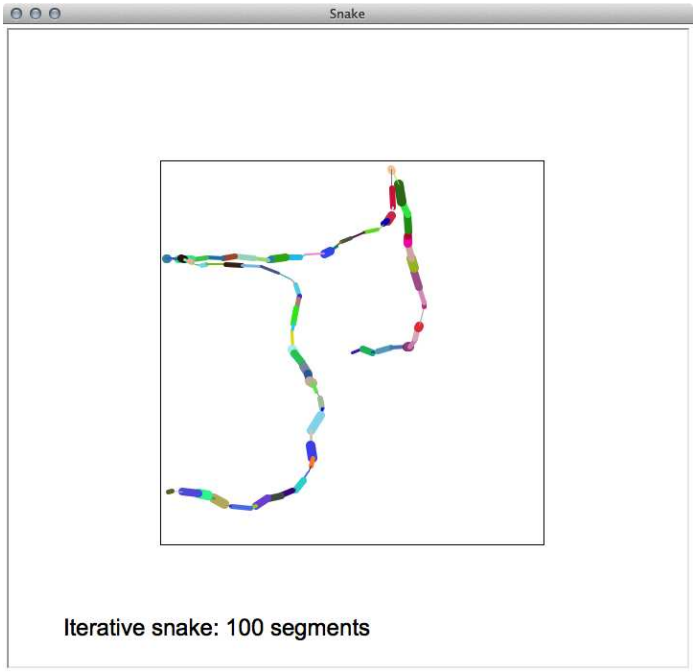
CSCI-141
Lab 04

09/09/2023

1 Problem

Using Python's turtle graphics module, design a program that draws snake-like figures in a randomly colorful fashion, based on a number of segments. When drawing, the snake must stay within the boundaries of the square outlined box.





1.1 Problem-solving Session (15%)

Work in a team of several students determined by the instructor. Each team will work together to complete the following activities. During problem-solving, do not consider color, thickness or any randomness. Also, do not worry about the bounding box until the last question.

1. Write a **non-tail-recursive** Python function, `draw_snake_rec`, that takes at least two arguments: a number that is the number of segments in the snake, and another number that is the starting length.
   At the end, the function returns the total length of the snake drawn. The visual effect in the turtle window is to draw a snake of the given number of segments.
   The function should draw a line segment of the given length, turn left 30 degrees, and then recurse reducing the length of the next segment by 1. Your code may not loop using any mechanism other than recursion.

2. Write a **tail recursive** Python function, `draw_snake_tail`, that takes two arguments as with the non-tail recursive function. To do this, you may decide to use additional argument(s) or write a *helper function*.
   At the end, the function returns the total length of the snake drawn. The visual effect in the turtle window is to draw a snake of the given number of segments.
   The function should draw a line segment of the current length, turn left 30 degrees, and then reduce the length of the next segment by 1. Your code may not loop using any mechanism other than recursion.

3. Draw a substitution trace for the call `draw_snake_rec(4, 10)`. (See the next page for an example.)

4. Write an iterative Python function `draw_snake_iter` that takes two arguments: a number that is the number of segments in the snake, and another number that is the starting length. The function must use a `while` loop, instead of recursion, and it should produce the same effect as the recursive version.

5. Create a **timeline** of changes to the parameters and local variables used as the call `draw_snake_iter(3, 5)` executes. (See the next page for an example.)

6. The program must keep the snake within the bounding box shown on the drawing canvas. Develop and describe a strategy for keeping the snake within that bounding box when the snake approaches a boundary from any random angle. **You can not use turtle.undo().**

At the end of problem-solving, the instructor might do a short discussion and one or more teams might be asked to present their work.

Substitution Traces

Recall from the lecture what a substitution trace looks like. It works by tracing the code as a sequence of substitutions of argument values as recursion progresses. Here's an example:

$$\begin{aligned}
\texttt{fact(3)} &= 3 * \texttt{fact(2)} \\
&= 3 * (2 * \texttt{fact(1)}) \\
&= 3 * (2 * (1 * \texttt{fact(0)})) \\
&= 3 * (2 * (1 * 1)) \\
&= 3 * (2 * 1) \\
&= 3 * 2 \\
&= 6
\end{aligned}$$

Timelines

A timeline shows the sequence of changes to variables as a block of code executes.

For this function:

```
def fact_iter( N ):
    value = 1
    while N > 1:
        value = N * value
        N -= 1
    return value
```

A timeline might look like this to show the state changes of the variables during loop execution:

N	value
3	1
2	3
1	6

1.2 In-lab Session (10%)

Begin work to *individually implement* a solution to this problem.

- Create and configure a folder or project for this assignment (e.g. lab04).
- Create the Python module `snake.py` to contain the solution.
- **Read the entire assignment to get a global view of the work.** Especially read the Turtle section on which functions you may and may not use.
- Start work on your design. The in-lab session outlines a **top-down** design approach in which you write stubs and make a main function call the stubs. Then, you fill in and test the stubs one by one until the program is complete.
  Eventually, your solution file should contain the following functions:

  1. Python implementations of the functions outlined in the problem-solving session (remove the length parameter since it will now be randomly generated). You can stub these initially.

  2. A `main` function that does the following (see the Requirements section for more information):

```
Prompt the user for the number of segments
If number of segments is out of range, print an error message and exit
Otherwise:
    Set up the drawing window for the recursive function
    Call the recursive drawing function and print the total length
    Pause and wait for the user to hit the enter key
    Reset the window for the iterative function
    Call the iterative drawing function and print the total length
    Print a message about closing the canvas window
    Wait for the user to close the canvas window
    End the program when the user closes the canvas
```

     You may use either the general recursive function or the tail recursive function for the recursive drawing.

  3. Other support or utility functions needed to promote function re-use (e.g. setting up the drawing window, drawing a single segment of the snake).

- If you have time before the In-lab deadline, decide how to create constants and use random functions to produce random effects. You can then add the imports and write utilities to support random effects. See the Constants and Random sections.

Zip your `snake.py` file into lab04.zip and upload this work-in-progress zip to the In-lab 04 box before the deadline in MyCourses.

2 Implementation (75%)

2.1 Requirements

- Assume you are using the default turtle drawing window. It is not required to label your drawings as recursive or iterative.
- The program should work in an IDE such as PyCharm, and also run from the command line as: `python3 snake.py`
- The program must prompt the user for the number of segments:
  `Segments (0-500):`
- If the number of segments is out of range, print the following message and exit:
  `Segments must be between 0 and  500 inclusive.`
- After each drawing is completed, print a message to standard output that displays the total length of the snake; for example:
  `The snake's length is #### units.`
- Pause the program between displaying the recursive and iterative snake.
- Use the same number of segments for each snake.
- You must display the bounding box and the snake must stay within that box for the entire drawing.

2.2 Console Output

Below is an example of the console output when the program runs; the pauses waiting for enter allow the user to view each drawing.

```
Segments (0-500): 123
Recursive snake's length is 1307 units.
Hit enter to continue...
Tail Recursive snake's length is 1295 units.
Hit enter to continue...
Iterative snake's length is 1305 units.
Close the canvas when finished viewing
```

2.3 Constants

Use the constant values below as **bounds on the range of a legal value**. Your code must check for a legal input that is in range. For values generated by the random number generator, your code must ensure that the value is also in the correct range.

Do not hardcode these as **magic numbers** in your code!

Ask your instructor and SLI for how to represent a constant in code; you might use global constant names or functions that return constants.

Names for constant values must be *fully capitalized* as shown.

- Your code prompts the user for the number of snake segments. The valid range for the number of segments is between `0` and `500` inclusive. Therefore `MAX_SEGMENTS` should be `500`.

- The bounding box should be between the points `(-200,-200)` and `(200,200)`. This is the drawing area the snake should stay within. That means `BOUNDING_BOX` should be `200`, the absolute value of the box's x or y value.
- `MAX_LENGTH` should be `20`. The maximum length of an individual segment of the snake should be between `1` and `20`.
- `MAX_THICKNESS` should be `10`. The maximum thickness corresponds to the pen size of an individual segment of the snake, and should be between `1` and `10`.
- `MAX_ANGLE` should be `30`. The maximum angle corresponds to the absolute value of the angle the turtle turns before drawing the next snake segment. The range of the angle turn should be between `-30` and `30`.

2.4 Random

There are two functions in the `random` module that will be useful:

- `randint(a,b)`: Returns a random integer in the range `[a,b]`, including the end-points. This function is useful for generating the random segment length, segment thickness and segment angle to turn.
  ```
  >>> random.randint(1,10)
  7
  ```
- `random()`: Returns a random floating point value in the range `[0,1]`, including the end points. This is useful for randomizing the color of each segment (see the Turtle section below for more information).
  ```
  >>> random.random()
  0.2128164618278381
  ```

2.5 Turtle

You will use a variety of functions from the turtle library to do your drawing:

- `pensize(size)`: Set the pen thickness to the given size.
- `pencolor(red, green, blue)`: By default, the `colormode` for turtle is `1`. This means that turtle expects values for each color parameter to be in the range of `[0,1]`. Think of this as the intensity of each color channel (0 is the lowest, 1 is the highest). Use the `random.random()` function to set each color channel value.
- `position()`: Return the position of the turtle on screen as an `(x,y)` tuple. This will be useful when dealing with the bounding box issue.
  For example, if the turtle's position is currently `(x,y) == (5,8)`, then the code fragment shows how you can assign the position values to the variables x and y.
  ```
  >>> x, y = turtle.position()
  >>> x
  5.0
  >>> y
  8.0
  ```
- `right(amount)` or `left(amount)`: turn the turtle by a fixed amount; a positive amount is counter-clockwise. Use this to randomly turn the snake for each segment.

- `setheading(angle)`: Change the turtle to face a different direction. The value `0` is East, `90` is North, `180` is West, `270` is South, and other values are in-between. This may be useful when dealing with the bounding box but is not required.
- `hideturtle()`: Hide the turtle so that it does not interfere with the drawing. You should hide the turtle before starting to draw.
- `reset()`: Reset the window by clearing everything and re-establishing the default state. Use this in between the drawings.
- `goto(x,y)` or `setpos(x,y)`: Use these only to return the turtle to the center of the screen *after drawing the bounding box*. After that, use only the `forward` or `back` functions to move the turtle.
- `done()`: Wait for a user to close the turtle window to end the program.

3 Grading (75%)

Your implementation grade is based on these factors:

- 20%: The program draws the snake recursively (15%), computes and returns its length (5%).
- 20%: The program draws the snake iteratively (15%) and computes its length (5%).
- 10%: The snake stays within the bounding box whenever drawn.
- 5%: The program prompts to get the number of segments and displays an error message and exits if the segment value is invalid.
- 5%: The program pauses between drawing the snakes and resets the turtle window before drawing the subsequent images.
- 5%: The program uses constant symbols and uses no magic numbers when drawing the snake.
- 10%: The code follows the documentation and style guidelines on the course web site, including docstrings for each function.

4 Submission

Zip your program file `snake.py` into a file called `lab04.zip` and upload the zip file to the **MyCourses dropbox** for this assignment before the due date on the box. Be sure to check that your upload completed successfully.