National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2019/2020

# Mission 2 - Side Quest
# Book of Advanced Spells

Release date: 28 August 2019
**Due: 04 September 2019, 23:59**

## Required Files

- sidequest02.1-template.py
- runes.py
- graphics.py
- PyGif.py

## Background

You chanced upon a shiny old book resting in one of the dark corners. Out of curiosity, you picked up the book and begin flipping through it. Most of the contents were basic spells that you have already learnt. However, you notice fine scribblings at the bottom of one page. Taking a closer look, you realise that it is a cool improvised spell. "Hmm.. some mage of old must have found her spellcasting class too easy," you think as you begin reciting the incantation.

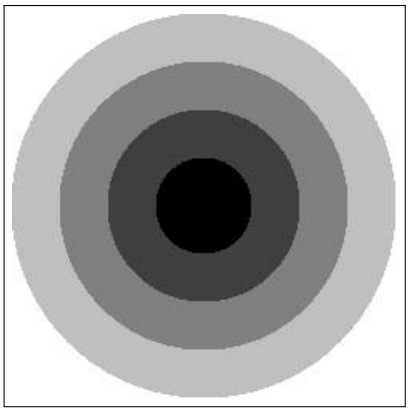
**Note**: Before you start on this side quest, you may want to read the background reading (see Appendix).

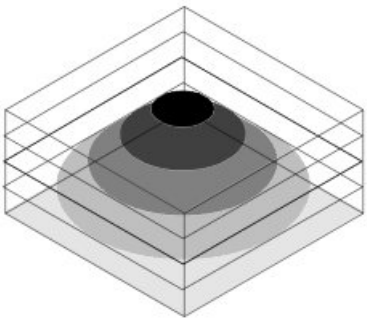
This side quest consists of **two** tasks.

## Task 1: Tree (5 marks)

Implement the function `tree` that takes a number of slices $n$ and a rune as arguments, and generates a stack of runes scaled and overlayed on top of each other.

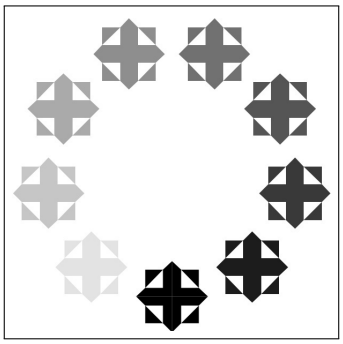
For example, the following command `show(tree(4, circle_bb))` should produce the following depth map:



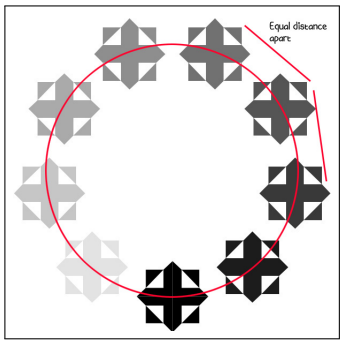


The generated tree must satisfy a few properties: the `circle_bb` at the top of the tree is scaled to $1/4$ of its original size (the tree has 4 layers); the next lower layer is scaled by $2/4$, and so on. Note that the bottom-most layer retains its original size. The different levels of the tree must also be spaced **evenly** apart.

## Task 2: Conjuring Helix (10 marks)



(a) show(helix(make_cross(rcross_bb),9))




(b) Illustrative diagram

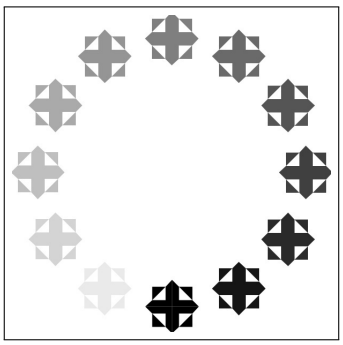
Figure 1: *(a) Helix with 9 runes, (b) Illustrative diagram*

Write a function `helix` that takes in a rune and number $n$ as arguments and creates runic patterns to form a helix, where $n \geq 5$. From the top-view, a helix is viewed as a pattern surrounded with a series of n runes spaced out **evenly** around forming a circle. From the side view, each rune is also spaced out **evenly** forming a 3D helix.[1]. The individual runes are scaled down to **2/n** of its original size and radius of the circle is **1/2 - 1/n**.

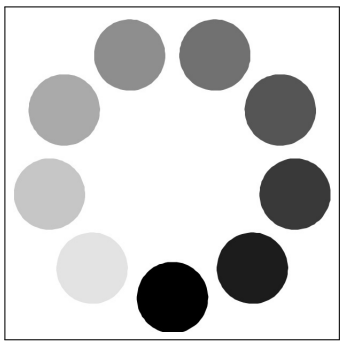
In particular, Figure 1 above can be created with:

```
show(helix(make_cross(rcross_bb),9))
```

[1] https://en.wikipedia.org/wiki/Helix



(a) show(helix(make_cross(rcross_bb),12))



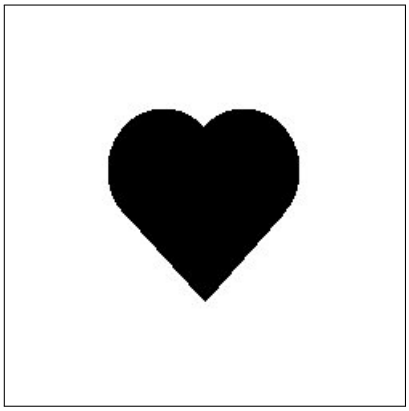
(b) show(helix(circle_bb,9))

Figure 2: *More examples. (a) is the same as previous example but with* $n = 12$*, (b) is the same as previous example but with the rune:* `circle_bb`

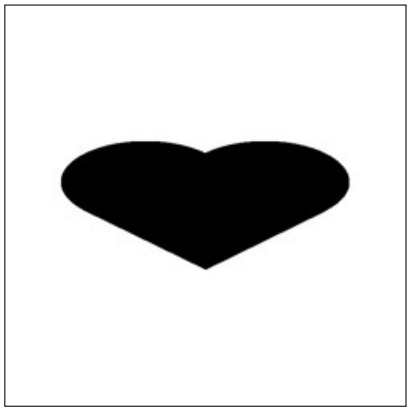
**Tip**: You will need to use the sine and cosine functions by importing the Python math module (`import math`) for this task. For example, to compute sin(45°), you can use: `math.sin(math.pi/4)` in your code.

## Appendix: Translation and Scaling

We have defined `scale` and `scale_independent`. The former scales the rune according to the ratio argument, the latter allows greater flexibility as you are able to pass two ratio arguments, one for the horizontal scaling while another is for vertical scaling.



(a) show(scale(1/2, heart_bb))



(b) show(scale_independent(3/4, 1/3, heart_bb))

Figure 3: *Examples of* `scale` *and* `scale_independent`. *In (a),* `heart_bb` *is scaled by* $1/2$ *both horizontally and vertically. In (b), it is scaled by* $3/4$ *horizontally and* $1/3$ *vertically.*

The last transformation tool we have is `translate`. The function translates the given rune by the given translation vector $(x, y)$ where $x$ and $y$ are ratios of a 600 x 600 viewport centred in the middle of the viewport (the actual size of the viewport is actually 800 x 600 so you will find that when a rune is translated by $1$ in the $x$-axis, it will still be visible whereas it will not when it is translated by $1$ in the $y$-axis.) Note that positive $x$ means translating to the right, while positive $y$ means translating down.

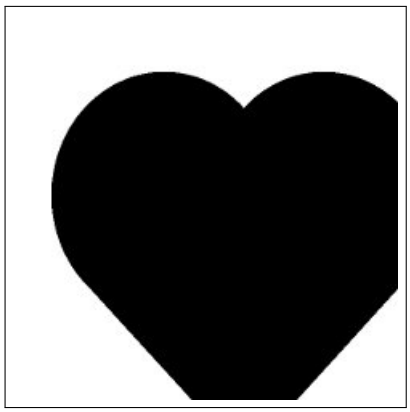


Figure 4: *Example of a translated* `heart_bb`. *This is the result of* `show(translate(0.1, 0.15, heart_bb))`

Try creating an anaglyph with the following command and view them with the 3D anaglyph glasses that was issued:

```
anaglyph(overlay(scale(0.8, heart_bb), circle_bb))
```

## Appendix: How to position runes on the viewport

This appendix will serve as a quick guide on how to position runes. Note that by default, all the runes are centered in the middle. Therefore, all translation operation should be **relative to the center of the viewport**.

**Positioning using x and y values**

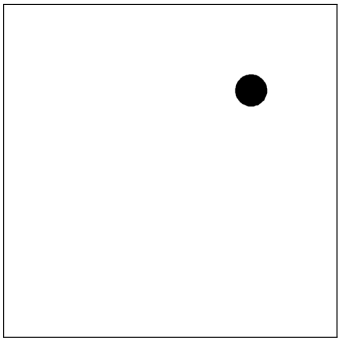


Figure 5: *Size reduced 90% and moved towards top-right.*

Code for moving a rune towards the top-right corner:

```
show(translate(0.25, -0.25, scale(0.1, circle_bb)))
```

**Note**: Translate is done with respect to a max unit of 1. i.e. A 0.5 translation will move a rune halfway across the viewport and 0.25 translation will move it one-quarter across the viewport.

**Positioning using angles**

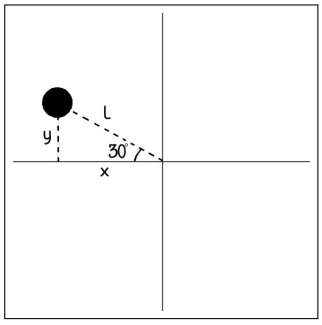



Figure 6: *Size reduced 90% and moved towards 30° direction.*

To find length $x$ on the x-axis, you can observe in the diagram that cos(30°) = $x/L$. Similarly, to find length $y$ on the y-axis, you get sin(30°) = $y/L$.

Therefore we can derive the equation:

```
x = L * cos(30°), y = L * sin(30°).
```

In the above example, we have $L$ = 0.4, $x$ = 0.346 and $y$ = 0.200. Therefore, we can obtain the above example with the code:

```
show(translate(-0.346, -0.2, scale(0.1, circle_bb)))
```

**Tip**: In the viewport, the y-axis is inverted.