National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2024/2025

Mission 1
Runic Paintings

Release date: 19th August 2024
Due: 1st September 2024, 23:59

General Restrictions

- No importing of additional packages (e.g. `math`). Necessary packages will be stipulated in the template file.
- Do not use any compound data structures, such as `tuple`, `list`, `dict`, `set`, etc.

Required Files

- mission01-template.py
- runes.py
- graphics.py
- PyGif.py

Background

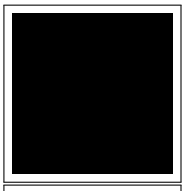
You are a newly enrolled wizard in the Python Institute for Mages (PIM) where only a small group of gifted humans are taught the sacred Python magic passed down from the ancient Python Gods. In this mission, you are expected to demonstrate your investigative skills. The ability to think and reason are necessary qualities to master Python magic.

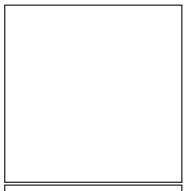
The grandmaster wizard has brought all your fresh wizards to the secret cave at the back of PIM, where you will need to open three doors, to lead you another step closer towards the inner sanctum of PIM.

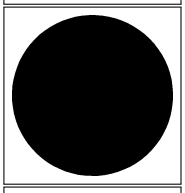
In the lecture, we demonstrated how Python can be used to generate runes. Now, you get to try your hand at drawing them. To do this, first open the file `runes.py` or `mission01-template.py` in IDLE and run the module. You need to ensure that all 4 files: `runes.py`, `graphics.py`, `PyGif.py`, `mission01-template.py` are in the **same** directory. A new window, called the viewport should appear. In `runes.py`, we defined the viewport (where your runes are going to be drawn) and the four runes discussed in class `rcross_bb`, `sail_bb`, `corner_bb`, and `nova_bb`. For example, you can display `rcross_bb` in the viewport with the command `show(rcross_bb)`.

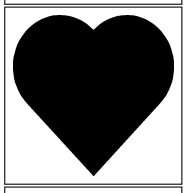
Don't forget to clear the viewport when necessary with `clear_all()`.

In addition to the ones you saw in lecture, we have also defined a few more basic runes that you can use:

`black_bb`: 

`blank_bb`: 

`circle_bb`: 

`heart_bb`: 

`ribbon_bb`: 

`pentagram_bb`: 

Note that all runes are squares. This applies to the inputs and outputs of all tasks in this mission.

In writing rather large, complex programs, one does not often understand (or even see) every single line of code, since such programs are usually written by several people, each in charge of smaller components. The key idea in functional abstraction is that as the programmer, you need not understand how each function works. All you need to know is what each function does and its signature (such as what parameters to pass). More specifically, **you need not read the code for these functions**.

You may refer to the lecture slides to understand what arguments each function requires and its corresponding effect. Now we will use these functions, defined in `runes.py`, as primitive building blocks, to draw our own pictures.

- `stack`
- `stack_frac`
- `quarter_turn_right`
- `eighth_turn_left`
- `flip_horiz`
- `flip_vert`
- `turn_upside_down`
- `quarter_turn_left`
- `beside`
- `make_cross`
- `repeat_pattern`
- `stackn`

This mission consists of **three** tasks.

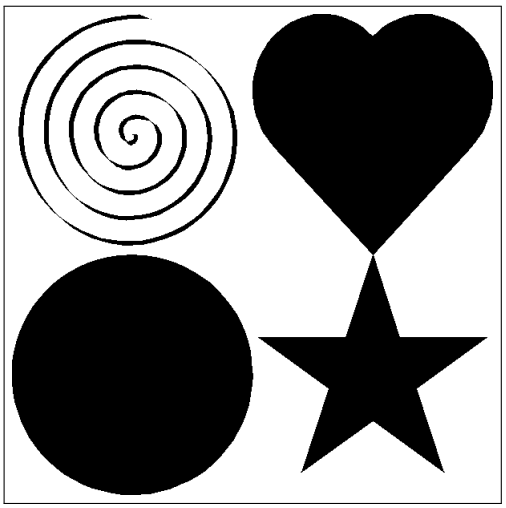
Task 1: Mosaic (4 marks)

On the first door you find 4 basic runes and one complex rune separated by an empty space. Clearly, your task would be to fill in this space with a descriptive function for the manipulation of the 4 basic runes to create the complex one.

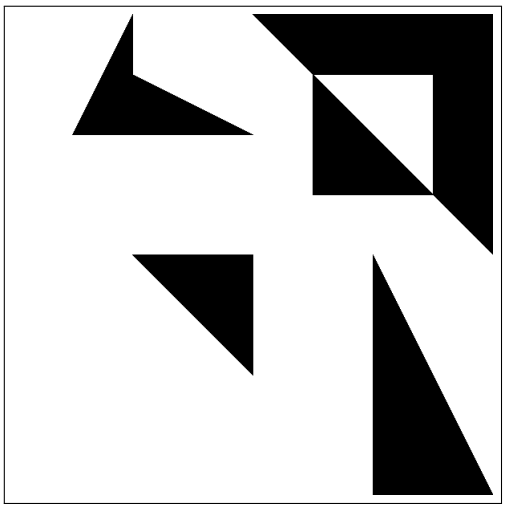
Write a function `mosaic` that takes as input four runes. `mosaic` **returns** (i.e. not `show`) a new rune from the four input runes. The four input runes are arranged in a 2×2 square **in equal proportions**, starting with the top-right corner, going clockwise.

Sample execution:

```
>>> mosaic_1 = mosaic(heart_bb, pentagram_bb, circle_bb, ribbon_bb)
>>> mosaic_2 = mosaic(rcross_bb, sail_bb, corner_bb, nova_bb)
```



(a) `show(mosaic_1)`



(b) `show(mosaic_2)`

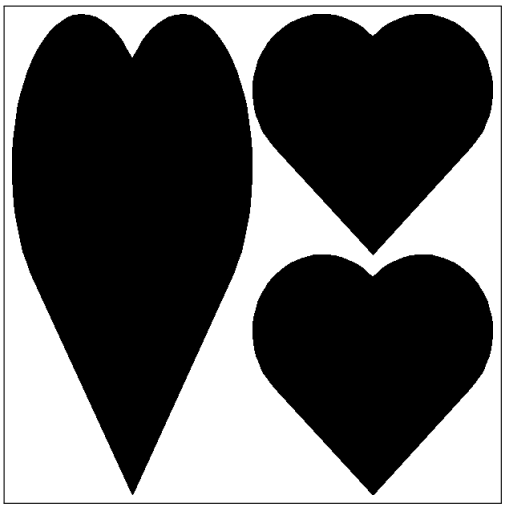
Task 2: Simple Fractal (4 marks)

On the second door, you find 2 runes displayed in a similar fashion. The only difference would be that the complex rune now exhibits a different layout.

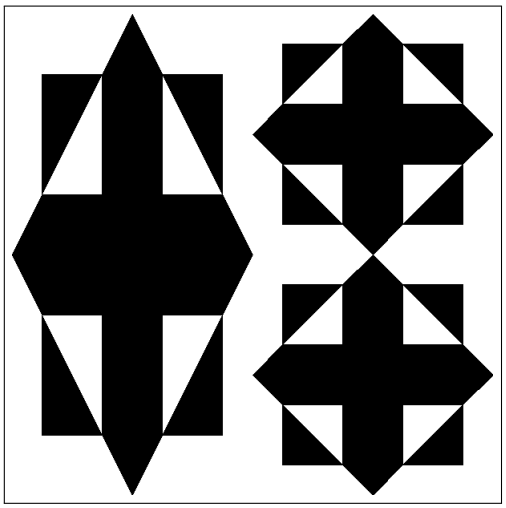
Write a function `simple_fractal` that takes as input a rune. `simple_fractal` **returns** a new rune composed from 3 copies of the input rune. The first copy is placed on the left, taking up **exactly half** the width of the final rune. The second and third copies are stacked on top of each other with **equal height**, and are placed on the right.

Sample execution:

```
>>> fractal_1 = simple_fractal(heart_bb)
>>> fractal_2 = simple_fractal(make_cross(rcross_bb))
```



(a) `show(fractal_1)`



(b) `show(fractal_2)`

More Background

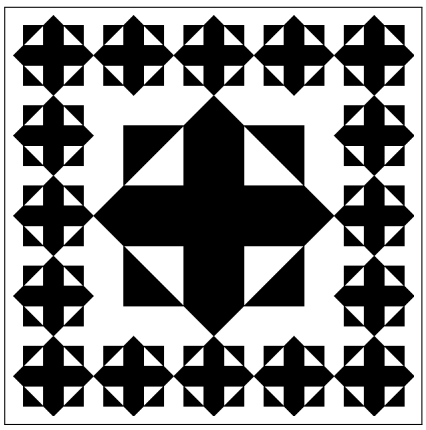
As you wander along the endless tunnels of the PIM cave, you notice several paintings along the wall. As you gazed upon the ornate Egyptian paintings adorning the palace walls, you discovered a fascinating pattern, a mesmerizing combination of repeating runic symbols elegantly arranged. The center, a captivating void filled with the same enchanting pattern, beckoned you to decipher its secrets. Upon questioning a trainer, you learn that these paintings are of Egyptian origin.

Then, the trainer introduced the legendary Egyptian artist, Zhu Ming, whose mastery of runic symbols and artistic expression transcends the ages. Recognizing your talent, Zhu Ming issues **Task 3** as your challenge.

Task 3: Egyptian (6 marks)

Note: If you find this problem too challenging, helper functions to aid you with this task will be introduced during the recitation.

Observing a nearest Egyptian painting, you notice that it exhibits **5** repeating patterns on every edge with the centre hollow filled with the same pattern:



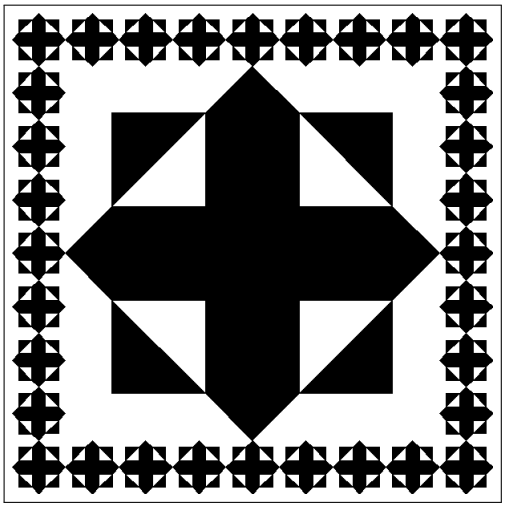
However, there are other Egyptian paintings that sport *different numbers* of repeating patterns. Your task is to write a function that enables Zhu Ming to create any of these paintings.

Write a function `egyptian` that takes as input a rune and an integer $n \geq 3$. `egyptian` **returns** a new rune composed from the input rune. n specifies the number of repetitions of **equally sized, square**, rune along all four **edges**, therefore, we have $n = 5$ for the above painting.

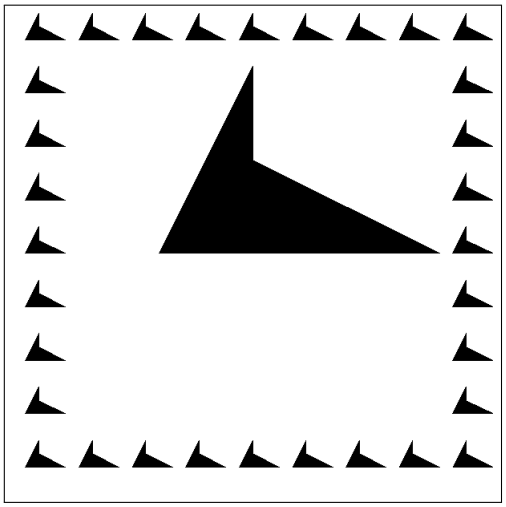
Note that the orientation of all images should remain the same as the input rune.

Consider the following Egyptian paintings and the command needed to recreate them:

```
>>> egyptian_1 = egyptian(make_cross(rcross_bb), 5) # As above
>>> egyptian_2 = egyptian(make_cross(rcross_bb), 9)
>>> egyptian_3 = egyptian(nova_bb, 9) # Orientation preserved
```



(a) `show(egyptian_2)`



(b) `show(egyptian_3)`