National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2022/2023

Mission 4
Curve Introduction

Release date: 4th September 2022
Due: 11th September 2022, 23:59

Required Files

- mission04-template.py
- hi_graph.py

Background

The grandmaster waves to the disciples as he ascends the stage. He introduces himself as Grandmaster Ben and begins to give a preparatory speech for the disciples.

> Welcome my disciples. I am impressed by your intelligence as I watched you solve the challenging rune problems of mosaic and fractals. However, those are just mere wand tricks. To become a proper wizard, you must learn to conjure magic from the mind. It will not be an easy feat. But we will start from scratch and learn the basics before advancing to manipulate reality.
>
> I will introduce to you some mind training. First, begin with an empty thought. Then slowly, picture a curve being drawn from one end to another. You will feel your mind strengthening as it draws out the lines of a curve...

Introduction

The objective of this mission is to introduce you to data abstraction. Through this mission, you will learn to work with objects that are not natively defined in Python, but are purely constructed by us. Data abstraction is important as it provides a way for you to work with functions without assuming the underlying implementation. We will be dealing with four primary types of objects in this mission:

Number This can be any kind of number, such as a float or an integer.

Unit-Interval This is a special kind of **Number**, restricted to between 0 and 1 inclusive.

Point This is an abstraction of a coordinate in 2D canvas.

Curve This is an abstraction of a curve drawn (with a pen) on a 2D canvas.

In addition, we will express the input and output types of a function in this way:

`<function>(`**`<parameter-type>`**`[, ...])` → **`<output-type>`**

Parameter-types denote the types of input parameters given to the function, while output-type denotes the type of output the function returns. You will see various examples of this in the mission.

Point Representation

A **Point** is a representation of a coordinate (x, y) in 2D coordinate space. A **Point** can only be created by a *constructor*, `make_point`, which constructs **Point**s from two **Number**s. You should always use `make_point` to produce a **Point** so as to not break abstraction. After a **Point** has been constructed, we can use *selectors*, `x_of` and `y_of`, to extract the x and y coordinate of the **Point** respectively. The format of the constructors and selectors is as follows:

```
make_point(Number, Number)  →  Point
      x_of(Point)           →  Number
      y_of(Point)           →  Number
```

Here is our implementation of **Point**:

```
def make_point(x, y):
    return lambda m: x if m == 0 else y

def x_of(point):
    return point(0)

def y_of(point):
    return point(1)
```

As an example, suppose we want to create a **Point**, `pt`, at $(0.5, 1.5)$. We would do the following:

```
pt = make_point(0.5, 1.5)
```

`pt` would then be the object type **Point**. Executing the following commands would give us the respective output:

```
>>> x_of(pt)
0.5
>>> y_of(pt)
1.5
```

Curve Representation

For this mission, **Curve**s are represented as *functions* that take in a **Unit-Interval** `t`, and return a **Point**. You can think of **Curve**s as **Point** generators. Intuitively, `t` can be thought of as the time, and the **Point** returned by the **Curve** can be thought of as the position of the pen at some specific time `t`. So, for example, at `t = 0`, the **Point** that is returned represents the *start* of the **Curve**. At `t = 1`, the **Point** that is returned represents the *end* of the **Curve**. At `t = 0.25`, the **Point** that is returned represents the position of the pen after the pen has moved through 25% of the **Curve**. The format of a generic **Curve** function is as follows:

```
Curve(Unit-Interval)  →  Point
```

For example, `unit_circle` is a **Curve** function that takes in a **Unit-Interval**, `t`, and returns a **Point** that lies on the circle centered about $(0, 0)$ with a radius of 1:

```
from math import *

def unit_circle(t):
    return make_point(sin(2*pi*t), cos(2*pi*t))
```

The starting point of the **Curve** function, `unit_circle`, can be obtained by calling `unit_circle(0)`, which returns the **Point** representing $(0, 1)$. The ending point is obtained using `unit_circle(1)`, which returns the **Point** representing $(0, 1)$. When `unit_circle(0.25)` is called, the **Point** along 25% of the `unit_circle`, $(1, 0)$, is returned.

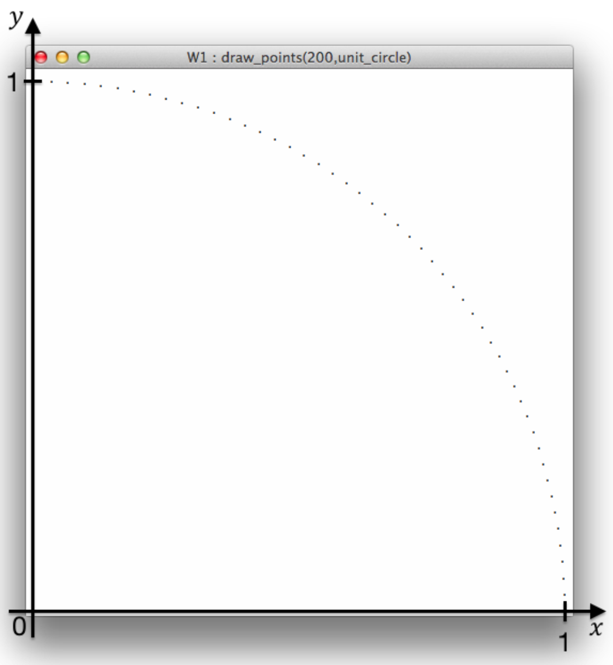
Displaying Curves

In the previous section, we have learnt the concept of a **Curve** as the basic drawing unit. In order to actually draw a **Curve** on the window, we will need a drawing function. It is not required that you understand the implementation of the various drawing functions, but you should know how to use them in order to visualize and test your solution.

Open **`hi_graph.py`** in IDLE, and run the file. In the Shell window, type:

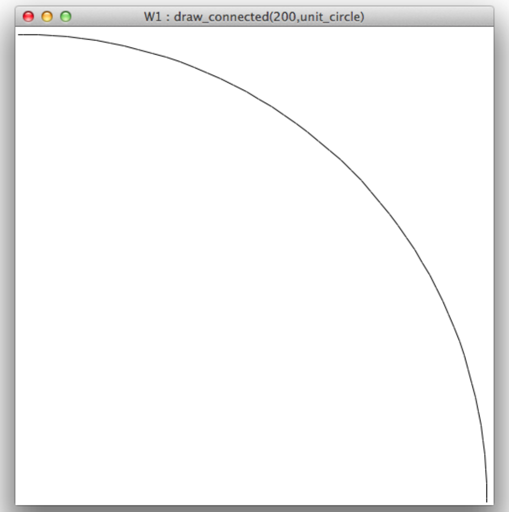
```
>>> draw_points(200, unit_circle) # dots along top-right of circle
>>> draw_points_scaled(200, unit_circle) # dots along circle
>>> draw_connected(200, unit_circle) # connected top-right of circle
>>> draw_connected_scaled(200, unit_circle) # connected circle
```

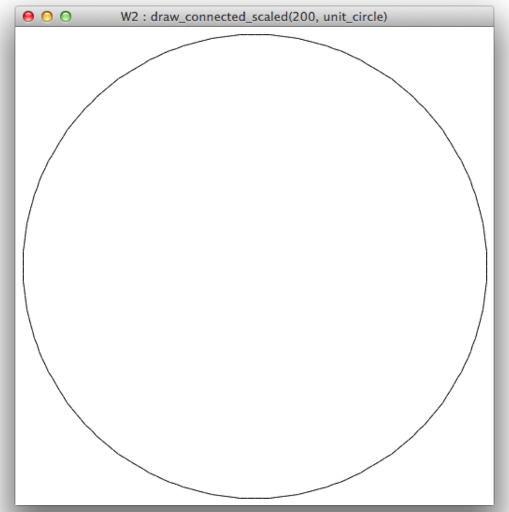
Note that after you have typed each line, a new window will appear. In the first window, you will see a quarter circle with separate dots. In the second window, you will see a full circle with dots. In the third window, a quarter circle is displayed. In the fourth window, you will see a full circle.



`draw_points(200, unit_circle)`

Figure 1: Displaying points without connecting them.





`draw_connected(200, unit_circle)` `draw_connected_scaled(200, unit_circle)`

Figure 2: Displaying connected points.

The value 200 in `draw_connected` refers to the number of points to draw in this screen. Since $1/200 = 0.005$, `unit_circle` will be called for values of $t = 0, 0.005, 0.010, 0.015, 0.020$, and so on, until $t = 1$. Note that the more points you use, the more accurately the curve will be drawn.

`draw_points` will draw points without connecting them. `draw_connected` will draw and sequentially connect every adjacent point to draw a connected **Curve**.

All of the drawing function type-signatures are shown here:

```
        draw_points(Number, Curve)  →  None
 draw_points_scaled(Number, Curve)  →  None
     draw_connected(Number, Curve)  →  None
draw_connected_scaled(Number, Curve)  →  None
```

The output of each draw function is **None** because the draw functions do not return any object. The draw functions merely render **Curve**s onto the drawing window, just like Python's `print` function, but specially for **Curve**s.

Note that the bottom-left corner of the drawing window represents the origin $(0, 0)$. Moving right along the drawing window increases the value of the x-axis until the x-coordinate equals 1. Likewise moving up along the window increases the value of the y-axis until the y-coordinate equals 1. You may notice `draw_connected` shows only a quadrant top-right of the `unit_circle`. This is because for some values of `t`, the x and y-coordinates are outside the range $[0, 1]$. This can be further checked by

```
>>> y_of(unit_circle(0.5))
-1.0
```

The y-coordinate of -1.0 is outside the range $[0, 1]$ and hence cannot be displayed. To display the full circle, use `draw_connected_scaled`, or `draw_points_scaled`. This function automatically scales and translates the **Curve** to make all points fall in the range $[0, 1]$ for both x and y-axes.

Your Mission

For your convenience, the template file **`mission04-template.py`** contains a line to load the Python source file **`hi_graph.py`**. Use the template file to answer the questions.

This mission has **two** tasks.

Task 1: (7 marks)

This is the definition of `unit_line_at_y` and `unit_line`:

```
def unit_line_at_y(y):
    return lambda t: make_point(t, y)

a_line = unit_line_at_y(0)
```

Answer the following questions.

Recall: Format to express a function's input and output types, together with an example, using the types described in the introduction, **Number**, **Unit-Interval**, **Point**, **Curve**:

```
<function>(<parameter-type>[, ...])  →  <output-type>
make_point(Number, Number)           →  Point
```

(a) What are the input and output types of the function `unit_line_at_y`?

(b) What are the input and output types of `a_line`?

(c) Define a function `vertical_line` with two arguments, a point and a length, that returns a vertical line of that length beginning at the point. Note that the line should be drawn upwards (i.e. towards the positive-y direction) from the point.

(d) What are the input and output types of `vertical_line`?

(e) Using `draw_connected` and your function `vertical_line` with suitable arguments, draw a vertical line which is centered both horizontally and vertically and has half the length of the sides of the window.

Task 2: (6 marks)

In addition to the direct construction of curves such as `unit_circle` or `unit_line` (already defined in **`hi_graph.py`**), we can use elementary Cartesian geometry in designing Python functions which *operate* on curves. For example, the mapping $(x, y) \longrightarrow (-y, x)$ rotates the curve by $\pi/2$ (anti-clockwise), so the following code

```
def rotate_90(curve):
    def rotated_curve(t):
        pt = curve(t)
        return make_point(-y_of(pt), x_of(pt))
    return rotated_curve
```

defines a Curve-Transform function which takes a **Curve** and transforms it into another, rotated **Curve**. The type of `rotate_90` is

```
rotate_90(Curve)  →  Curve
```

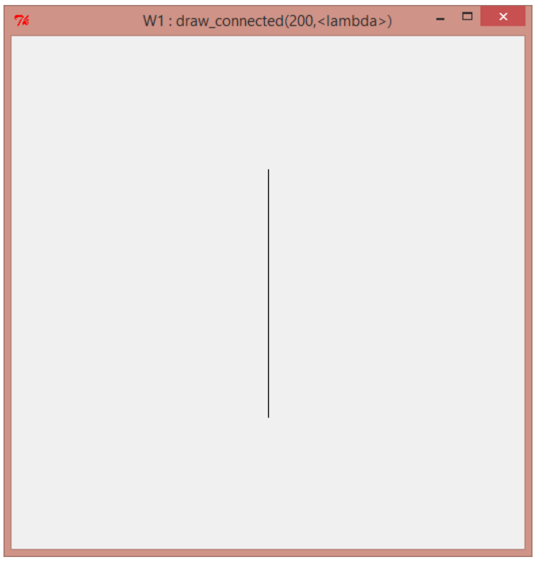


Figure 3: A centered vertical line

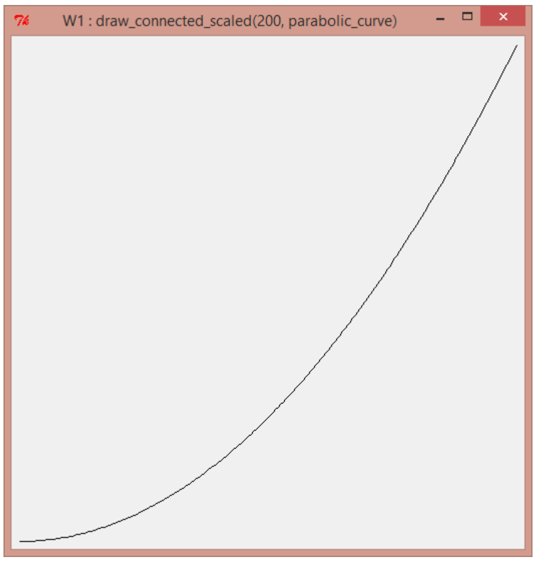


Figure 4: The original curve

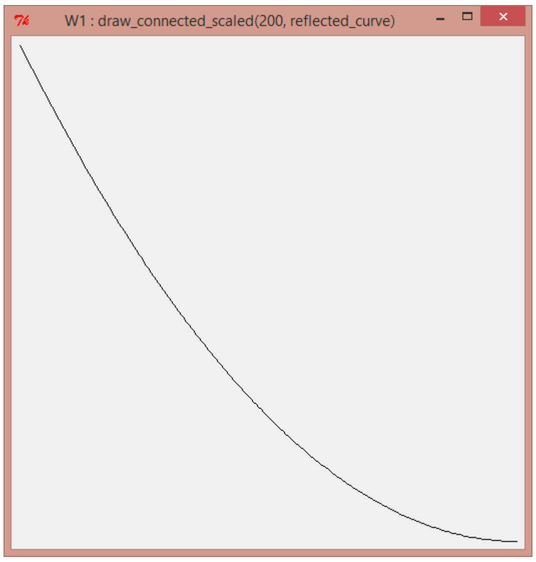


Figure 5: The reflected curve

We would like to create a new Curve-Transform `reflect_through_y_axis`, which turns a curve into its mirror image. An example of this transformation is provided above.

(a) Briefly describe how you would test `reflect_through_y_axis` to verify it works properly.

(b) Write your definition of Curve-Transform `reflect_through_y_axis`.

Note:

- It is actually fine if the curve reflects in the y-axis and disappears from the viewport. To view the effect and the curve in the viewport, you may wish to try `draw_points_scaled` or `draw_connected_scaled`.