National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2021/2022

Mission 4
Curve Introduction

Release date: 07 February 2022
Due: 25 February 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 problems of the rune doors and mosaic. However, these 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 I will start from scratch and teach you the basics before you advance to manipulate reality.
>
> I will introduce to you some mind training. You will begin to think of nothing first. Then slowly, start picturing a curve being drawn from one end to another. You will feel your mind strengthening as it codes out the shape of the curve...

Introduction

The objective of this mission is to introduce you to data abstraction. Through this mission, you should be able 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 knowing how they are implemented. We will be dealing with three primary types of objects in this mission:

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

Point This is an abstraction of a coordinate in 2D space, and can be constructed with `make_point`. You should always use `make_point` to produce a Point.

Curve This is an abstraction of a line. It is, in essence, a collection of Points. Later sections will elaborate further on this.

In addition, we will be introducing a format for expressing a function's type, as follows:

$$\texttt{<function-name>} \;:\; (\text{<parameter-type>}[, \ldots]) \rightarrow \text{<output-type>}$$

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

Point

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 Points from Numbers. 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. The format of the constructors and selectors is as follows:

$$\begin{aligned} \texttt{make\_point} &: (\text{Number}, \text{Number}) \rightarrow \text{Point}, \\ \texttt{x\_of} &: (\text{Point}) \rightarrow \text{Number}, \\ \texttt{y\_of} &: (\text{Point}) \rightarrow \text{Number}. \end{aligned}$$

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 $(0.5, 0.5)$. We would do the following:

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

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)
0.5
```

Curve Function

In order to draw curves, however, we will need a collection of points. The Curves in this mission are **functions** that take in a Number, t, which will fall within the unit interval $[0, 1]$, and return a Point. You can think of Curves 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 the time indicated by the argument. We let Unit-Interval be the type of Number between 0 and 1. So, for example, at $t = 0.25$, the Point that is produced is where the pen is after it has moved through 25% of the Curve. The format of a Curve function is as follows:

$$\text{Curve} : (\text{Unit-Interval}) \rightarrow \text{Point}$$

For example, `unit_circle` is a Curve function that takes in a Unit-Interval t, and returns a Point:

```
from math import *

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

cir = unit_circle
```

The starting point of cir can be obtained by calling `cir(0)`, while the ending point is returned by `cir(1)`. (Refer to the intuitive understanding of t as time.)

Displaying Curves

In the previous section, we have learnt the concept of 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 drawing functions, but you should know how to use them in order to visualize and test your solution.

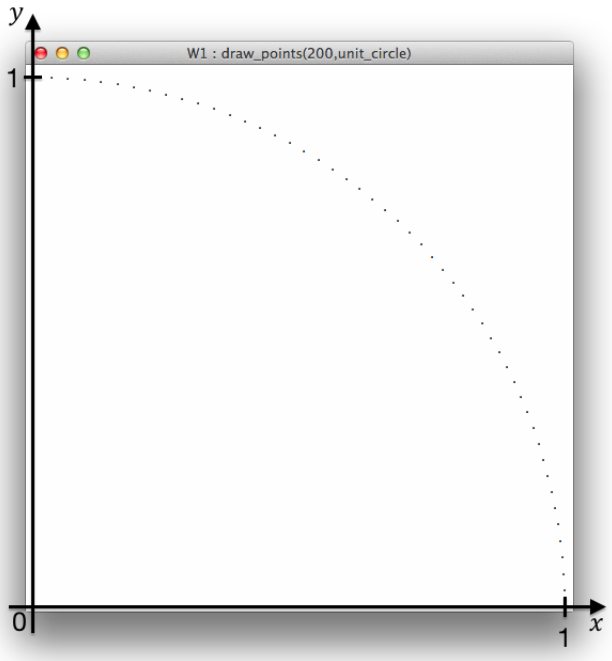
Open **`hi_graph.py`** in IDLE, and run the file. In the Shell window, type:

```
>>> draw_points(200, unit_circle)
>>> draw_points_scaled(200, unit_circle)
>>> draw_connected(200, unit_circle)
>>> draw_connected_scaled(200, unit_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.

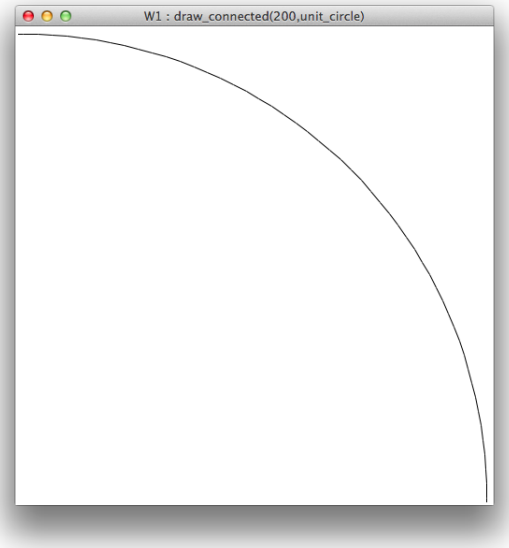
The value 200 in `draw_connected` refers to the number of points to draw in this screen. Since $1/200 = 0.005$, `unit_circle` will be called for values of $t = 0, 0.005, 0.01, 0.015, 0.02$, and so on, until $t = 1$. Note that the more points you use, the more accurately the curve will be drawn.

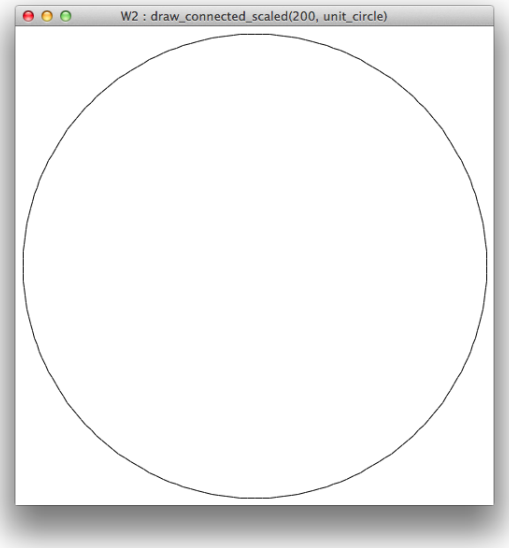
`draw_points` will draw points without connecting them. `draw_connected` will then join two adjacent points returned by `unit_circle` to draw a connected Curve.



`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.

All of the drawing methods (ie, `draw_points`, `draw_connected` etc) are of this form:

$$(\text{Number}, \text{Curve}) \rightarrow \text{Empty}$$

(The output of the draw functions is Empty because the draw functions do not return any object. They merely render Curves onto the drawing window.)

Note that the origin of the drawing window is at the bottom left. 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 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))
```

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 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.

(a) What is the type of the input and output of the function `unit_line_at_y`?
Recall: The format for expressing the input and output types, and an example, are shown below:

$$\begin{aligned} \texttt{<func>} &: (\texttt{<para-type>}[, \ldots]) \rightarrow \texttt{<output-type>} \\ \texttt{make\_point} &: (\text{Number}, \text{Number}) \rightarrow \text{Point} \end{aligned}$$

(b) What is the type of the input and output 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 is the type of the input and output 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.

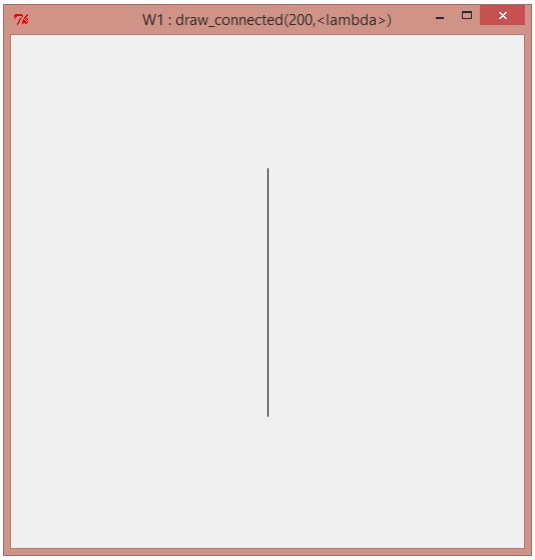


Figure 3: A centered vertical line

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 function which takes a curve and transforms it into another, rotated, curve. The type of `rotate_90` is

$$\texttt{rotate\_90} : (\text{Curve}) \to \text{Curve}.$$

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 below.

(a) Briefly describe how you would check that the curve transformation 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 might try `draw_points_scaled` or `draw_connected_scaled`.

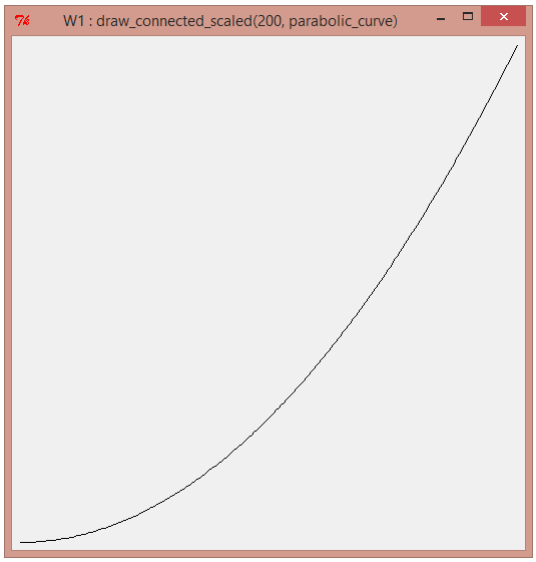


Figure 4: The original curve

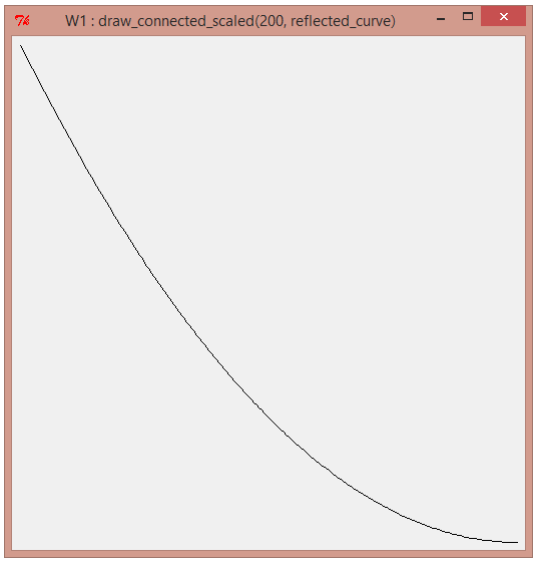


Figure 5: The reflected curve