

Assessment 2 Brief	
Subject Code and Title	ITP122 Introduction to Programming
Assessment	Intermediate Programming Tasks
Individual/Group	Group
Length	Source code and comments
Learning Outcomes	The Subject Learning Outcomes demonstrated by successful completion of the task below include: a) Select appropriate programming development tools and methodologies to meet the software requirements. b) Apply coding, debugging and testing skills in software development using a suitable IDE (integrated development environment) platform. c) Validate and verify computer programs to meet the software requirements.
Submission	Due by 11:55pm AEST/AEDT, Sunday, end of Module 08
Weighting	45%
Total Marks	100 marks

Assessment Task

You must complete **programming tasks** that demonstrate your understanding of decision logic, including (`if-else` statements) and loops (`for` and `while`), based on the concepts covered in Modules 2 to 8. Each program should include clear and concise documentation in the form of comments explaining the logic and functionality of the code.

Context

This assessment focuses on writing Python scripts that generate graphic output using the **Thonny Python IDE** with the **Thonny-py5mode plugin**. To set up your development environment for the assessment, watch the relevant installation guide video:

- [View Windows video guide](https://github.com/villares/thonny-portable-with-py5/releases/latest) (software: <https://github.com/villares/thonny-portable-with-py5/releases/latest>)
- [View macOS & Linux video guide](https://thonny.org) (software: <https://thonny.org>)

In Assessment 1, you applied and explained the use of variables, expressions, basic and intermediate conditionals, and other fundamental programming concepts. In this assessment, you'll expand upon

this knowledge by implementing intermediate-level decision logic and loops, marking an important step toward developing problem-solving skills and confidence in devising algorithmic solutions.

Key Objectives:

You're required to submit all the programming tasks for this Assessment:

- Implementing `if-else` statements for decision-making
- Using `for` and `while` loops to manage repetition
- Generating **graphic output** using the **Thonny-py5mode plugin**
- Providing well-structured comments to explain your code

Instructions

Each task begins with some code to get you started. The goal is to complete the script so that your output matches each graphic provided. Here's a summary of the different functions you'll need:

Fills & Strokes

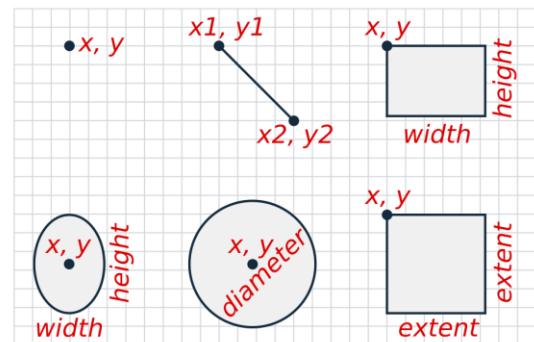
```
background(color) # set bg color
fill(color) # set fill color
no_fill() # disable the fill
stroke(color) # set stroke color
stroke_weight() # stroke width in px
no_stroke() # disable the stroke
```

A red fill using three different color values:

```
fill('#FF0000') # hexadecimal
fill(255, 0, 0) # red, green, blue
# HSB (Hue Saturation brightness)
color_mode(HSB, 360, 100, 100)
fill(0, 100, 100)
```

2D Primitives

```
point(x, y)
line(x1, y1, x2, y2)
rect(x, y, width, height)
ellipse(x, y, width, height)
circle(x, y, diameter)
square(x, y, extent)
```



Note that y-coordinates increase as you move downward, while x-coordinates increase as you move rightward. In other words, origin -- the coordinate (0, 0) -- is the top-left of the display window.

You'll add Python `if`, `else`, `for`, and other statements to those above to successfully complete each task.

A Quick Demonstration

Using Thonny-py5mode, you can draw lines and shapes in various colours using different functions. Here's a basic, commented example:

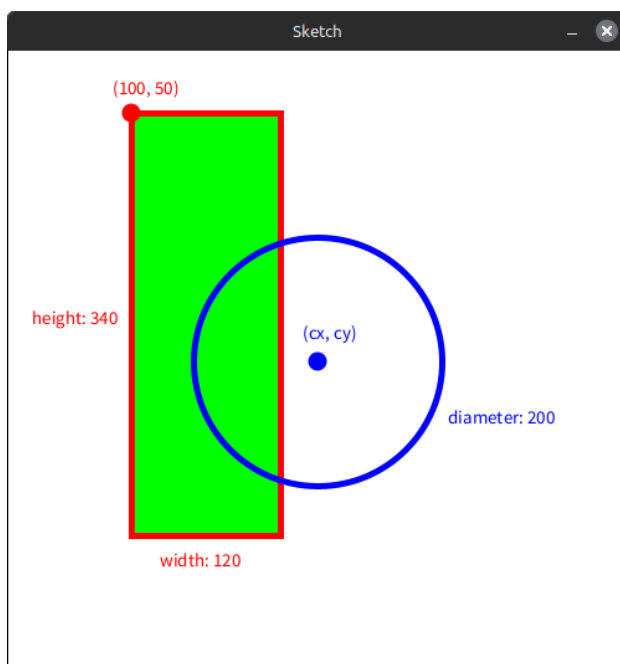
```
# setup
size(500, 500)           # canvas size
background('#FFFFFF')    # white background colour
cx = width / 2           # canvas horizontal centre
cy = height / 2          # canvas vertical centre
stroke_weight(5)         # set outline to 5-pixels-wide

# draw rectangle
stroke('#FF0000')        # set outline to red
fill('#00FF00')          # set fill to green
rect(100, 50, 120, 340)  # draw rectangle

# draw circle
no_fill()                # set fill to none
stroke('#0000FF')        # set outline to blue
circle(cx, cy, 200)      # draw circle
```

Note that commands like `stroke()` and `fill()` remain in effect until you override their behaviour, like using a paint brush -- when you dip the brush in a red pot, it'll paint in red shapes, until you dip it in blue. You'll notice that colours are specified using *hexadecimal* values. You can access a hexadecimal mixer using **py5 > Color selector** in the Thonny interface.

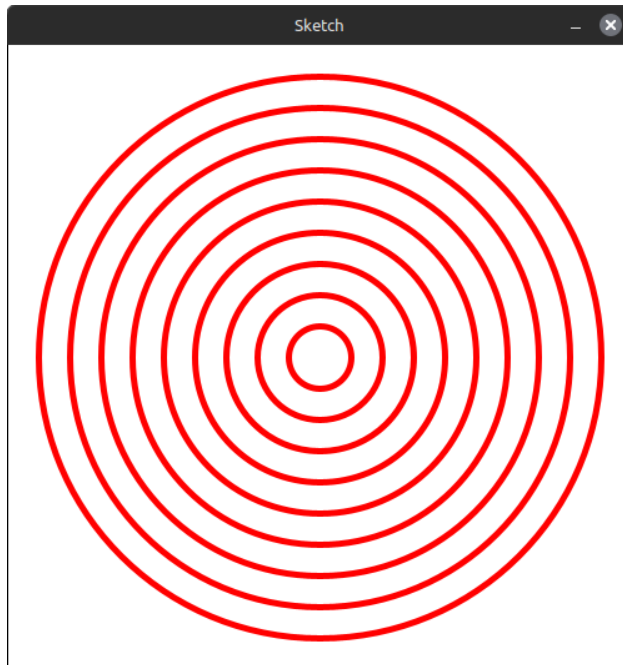
Here's the result, labelled so you can tell how the dimensions correspond to function arguments:



Task 1

Task 1.1

Replicate this result as closely as you can using different Python techniques and drawing functions:



Module 6 (*Simple Loops*) covers the techniques you'll need to employ here.

Starter code:

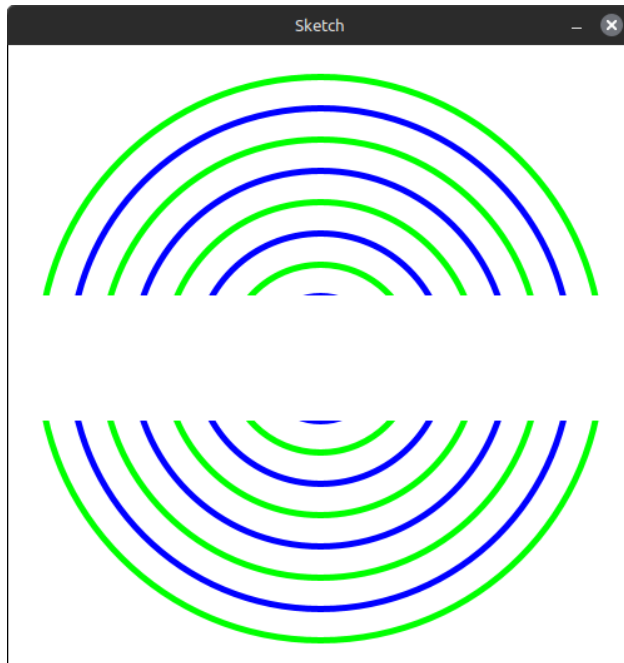
```
# setup
size(500, 500)           # canvas size
background('#FFFFFF')    # white background colour
cx = width / 2           # canvas horizontal centre
cy = height / 2          # canvas vertical centre

i = 1

while i < 10:
    no_fill()
    stroke_weight(5)
    # ... INSERT MISSING CODE HERE ...
    i += 1
```

Task 1.2

Replicate this result as closely as you can using different Python techniques and drawing functions:



Consider how you might nest conditional statements within a loop (see *Module 8: Intermediate Loops*), and how you can visually mask lines by drawing shapes over them (filled the same colour as the background).

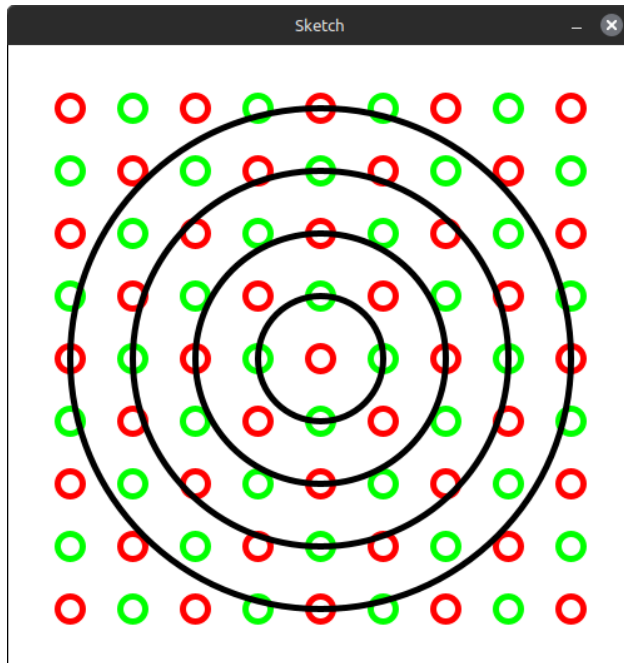
Starter code:

```
# setup
size(500, 500)           # canvas size
background('#FFFFFF')    # white background colour
cx = width / 2           # canvas horizontal centre
cy = height / 2          # canvas vertical centre
no_fill()                # set fill to none
stroke_weight(5)          # set outline to 5-pixels-wide

# ... INSERT MISSING CODE HERE ...
```

Task 1.3

Replicate this result as closely as you can using different Python techniques and drawing functions:



Consider how you might use loop within a loop to achieve a grid-like arrangement of shapes. Modules 5 and 8 cover *Nested Decision Logics* and *Intermediate Loops*, respectively.

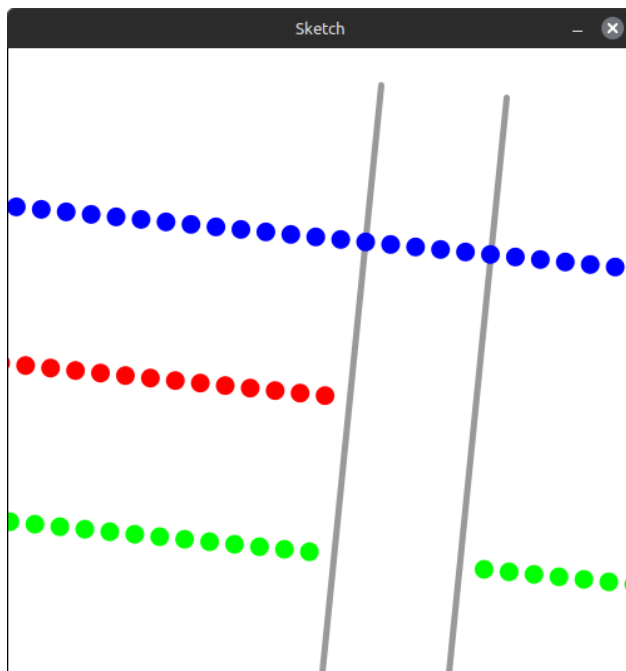
Starter code:

```
size(500, 500)
background('#FFFFFF')

# ... INSERT MISSING CODE HERE ...
```

Task 2

Replicate this result as closely as you can using different Python techniques and drawing functions:



Use (for?) loops to draw the rows of blue, red, and green dots. You must use `break` and/or `continue` statements (see *Module 8: Intermediate Loops*) to control the interruptions of the red and green dots. **TIP:** Consider nesting `if` statements within your loops and how you might utilise the `line_1_x` and `line_2_x` variables.

Starter code:

```
size(500, 500)
background('#FFFFFF')
rotate(0.1) # rotates the entire drawing space clockwise by ~6°

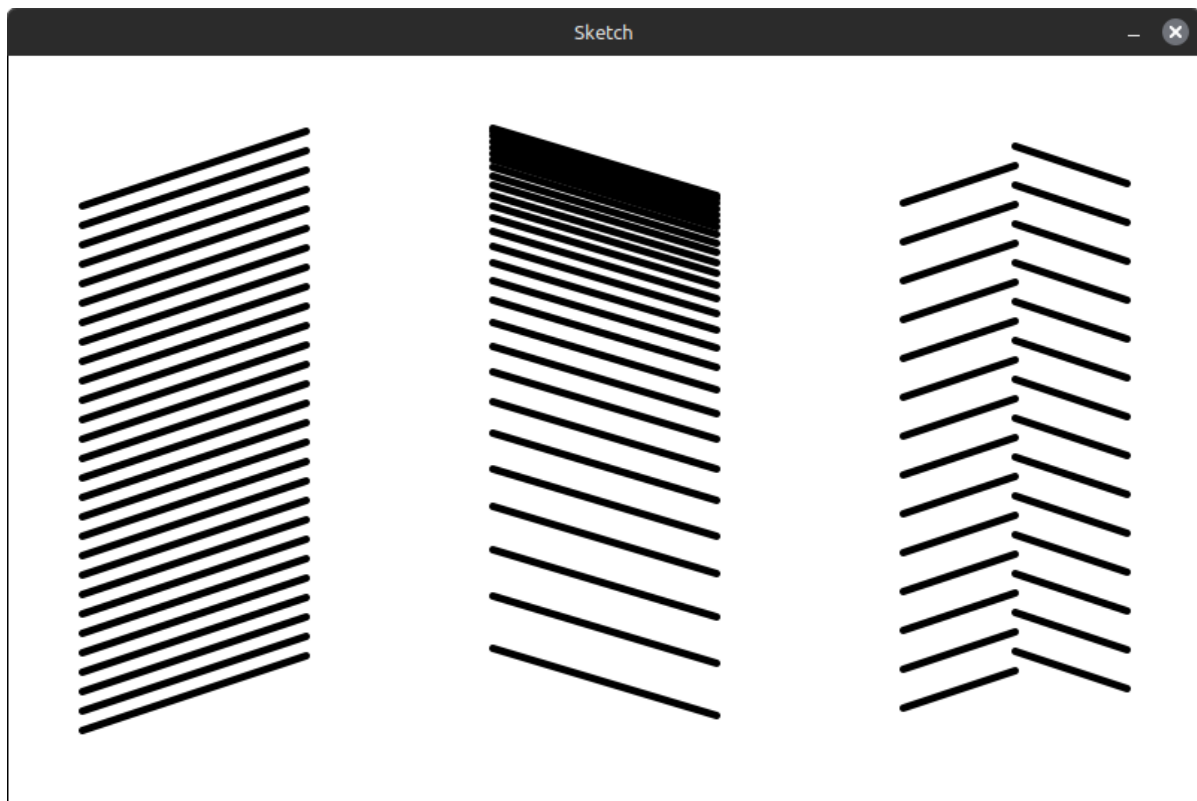
line_1_x = 300
line_2_x = 400

# draw two grey lines
stroke_weight(5)
stroke('#999999') # grey
line(300, 0, line_1_x, height)
line(400, 0, line_2_x, height)
no_stroke()

# loop for blue dots
# loop for red dots
# loop for green dots
```

Task 3

Replicate this result as closely as you can using different Python techniques and drawing functions:



Draw all three patterns in a single canvas, using a separate loop for each. You'll need to use the `line()` function. You may find the `translate()` function useful:

<https://py.processing.org/reference/translate> (although this isn't required).

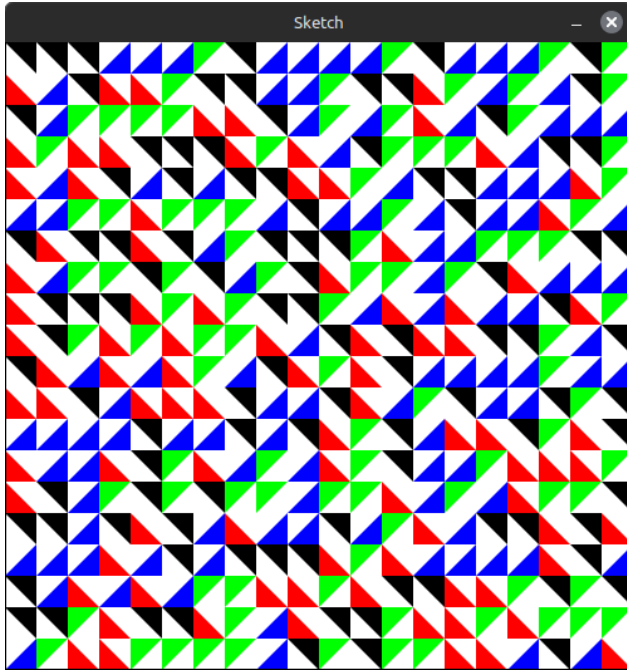
Starter code:

```
size(800, 500)
background('#FFFFFF')
stroke('#000000')
stroke_weight(5)

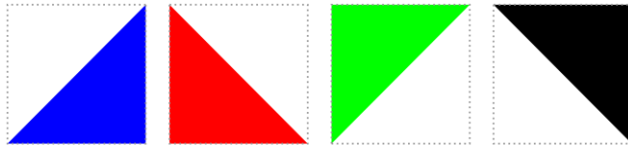
# ... INSERT MISSING CODE HERE ...
```

Task 4

This task involves **Truchet tiles** (https://en.wikipedia.org/wiki/Truchet_tiles) -- a set of four contrasting tiles randomly arranged in a grid formation to make interesting patterns.



Here are the four tiles, which you can draw using the `triangle()` function:



Starter code:

```
size(500, 500)
background('#FFFFFF')
no_stroke()
tile_size = 25

# single blue tile demonstration
fill('#0000FF')
triangle(
    tile_size, 0,           # point 1 x-y coord
    tile_size, tile_size,  # point 2 x-y coord
    0, tile_size           # point 3 x-y coord
)
```

(continued on next page)

You'll need to use the `random()` function, which returns a random value between 0.0 and 0.1

```
# ten random values between 0.0 and 1.0
for i in range(10):
    print(random())
```

As an example, the above code might produce:

```
0.12961569776802295
0.520657584472319
0.2004921567441561
0.7942788165062895
0.35182071797307446
0.46077237172680174
0.5828424497881715
0.911555329559598
0.8145882527706059
0.030076611396973485
```

Submission

Prepare your files

- Ensure you have written your student ID and name as a comment at the top of each file
- Make a **zip** folder containing all your files (namely, a Python file for each of the following tasks: 1.1, 1.2, 1.3, 2, 3, and 4)
- Name your zip file to match this convention: *ITP122_LastnameFirstname_A2.zip*

Credit any sources

As many online programming resources provide solutions to programming tasks along with documentation, if any such source code is acquired (reference works, documentation, help and tutorial sites, etc), it must be preceded by a code comment that lists the original site/creator and followed by a comment that declares the end of the acquired code. Acquisitions should be kept to a few lines or less and solve single problems (i.e., changing the range of a randomly generated number, using additional drawing functions you may have researched, and so forth).

Submit your files

Your submission will contain the zip archive of your project; submit this via the **Assessment 2 - Submission** link, accessible via the **Briefs & Submissions** link within main navigation menu of the *ITP122 Introduction to Programming* MyLearn portal.

Your Lecturer(s) will provide grades and feedback via MyLearn.

Before you submit your assessment, please ensure you have read and understood Torrens Academic Integrity policies: https://library.torrens.edu.au/academic_integrity. If you are unsure about anything, please reach out to your lecturer.

Assessment Attributes	Fail (Yet to achieve minimum) 0–49%	Pass (Functional) 50–64%	Credit (Proficient) 65–74%	Distinction (Advanced) 75–84%	High Distinction (Exceptional) 85–100%
Programming Task 1: Demonstrated understanding of the task and applied knowledge in programming the given task/problem. Percentage for this criterion = 25%	Coding/implementation of the task produces large differences between the original and coded graphic. Poor code layout, commenting, and variable naming. Generally, a poor effort, and the program fails to run.	Coding/implementation of the task produces easily discernible differences between the original and coded graphic. Acceptable code layout, commenting, and meaningful variable names. Program runs, but with obvious room for code improvements.	Coding/implementation of the task produces some discernible differences between the original and coded graphic. Sensible code layout, commenting, and meaningful variable names. Program runs, but there is some room for improvement to enhance the accuracy of the replicated graphic.	Coding/implementation of the task produces minor discernible differences between the original and coded graphic. Good code layout, commenting, and meaningful variable names. Program runs well, exhibiting a solid grasp of Python programming concepts.	Coding/implementation of the task produces a highly accurate recreation of the original graphic. Code demonstrates best practices regarding layout, comments, variable names. Program runs very well, exhibiting a solid grasp of coding techniques in line with- and slightly beyond those covered in classes.
Programming Task 2: Demonstrated understanding of the task and applied knowledge in programming the given task/problem. Percentage for this criterion = 25%	Coding/implementation of the task produces large differences between the original and coded graphic. Poor code layout, commenting, and variable naming. Generally, a poor effort, and the program fails to run.	Coding/implementation of the task produces easily discernible differences between the original and coded graphic. Acceptable code layout, commenting, and meaningful variable names. Program runs, but with obvious room for code improvements.	Coding/implementation of the task produces some discernible differences between the original and coded graphic. Sensible code layout, commenting, and meaningful variable names. Program runs, but there is some room for improvement to enhance the accuracy of the replicated graphic.	Coding/implementation of the task produces minor discernible differences between the original and coded graphic. Good code layout, commenting, and meaningful variable names. Program runs well, exhibiting a solid grasp of Python programming concepts.	Coding/implementation of the task produces a highly accurate recreation of the original graphic. Code demonstrates best practices regarding layout, comments, variable names. Program runs very well, exhibiting a solid grasp of coding techniques in line with- and slightly beyond those covered in classes.
Programming Task 3: Demonstrated understanding of the task and applied knowledge in	Coding/implementation of the task produces large differences between the original and coded graphic.	Coding/implementation of the task produces easily discernible differences between the original and coded graphic.	Coding/implementation of the task produces some discernible differences between the original and coded graphic.	Coding/implementation of the task produces minor discernible differences between the original and coded graphic.	Coding/implementation of the task produces a highly accurate recreation of the original graphic.

programming the given task/problem. Percentage for this criterion = 25%	Poor code layout, commenting, and variable naming. Generally, a poor effort, and the program fails to run.	Acceptable code layout, commenting, and meaningful variable names. Program runs, but with obvious room for code improvements.	Sensible code layout, commenting, and meaningful variable names. Program runs, but there is some room for improvement to enhance the accuracy of the replicated graphic.	Good code layout, commenting, and meaningful variable names. Program runs well, exhibiting a solid grasp of Python programming concepts.	Code demonstrates best practices regarding layout, comments, variable names. Program runs very well, exhibiting a solid grasp of coding techniques in line with- and slightly beyond those covered in classes.
Programming Task 4: Demonstrated understanding of the task and applied knowledge in programming the given task/problem. Percentage for this criterion = 25%	Coding/implementation of the task produces large differences between the original and coded graphic. Poor code layout, commenting, and variable naming. Generally, a poor effort, and the program fails to run.	Coding/implementation of the task produces easily discernible differences between the original and coded graphic. Acceptable code layout, commenting, and meaningful variable names. Program runs, but with obvious room for code improvements.	Coding/implementation of the task produces some discernible differences between the original and coded graphic. Sensible code layout, commenting, and meaningful variable names. Program runs, but there is some room for improvement to enhance the accuracy of the replicated graphic.	Coding/implementation of the task produces minor discernible differences between the original and coded graphic. Good code layout, commenting, and meaningful variable names. Program runs well, exhibiting a solid grasp of Python programming concepts.	Coding/implementation of the task produces a highly accurate recreation of the original graphic. Code demonstrates best practices regarding layout, comments, variable names. Program runs very well, exhibiting a solid grasp of coding techniques in line with- and slightly beyond those covered in classes.