Visualization using Python

An introduction to Matplotlib

Benoît Corsini

March 16th and 30th 2026

Disclaimers

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

https://www.benoitcorsini.com/files/matplotlib/q0.py

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

https://www.benoitcorsini.com/files/matplotlib/q0.py

- This presentation is composed of several coding and visual questions.

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

https://www.benoitcorsini.com/files/matplotlib/q0.py

- This presentation is composed of several coding and visual questions.
  - A "correct" answer is not a word for word file or a pixel for pixel image, but rather a method that provides a similar outcome.

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

https://www.benoitcorsini.com/files/matplotlib/q0.py

- This presentation is composed of several coding and visual questions.
  - A "correct" answer is not a word for word file or a pixel for pixel image, but rather a method that provides a similar outcome.
  - The solution of the different questions are available online, by replacing the 0 from the above link with the corresponding question number.

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

https://www.benoitcorsini.com/files/matplotlib/q0.py

- This presentation is composed of several coding and visual questions.
  - A "correct" answer is not a word for word file or a pixel for pixel image, but rather a method that provides a similar outcome.
  - The solution of the different questions are available online, by replacing the 0 from the above link with the corresponding question number.
  - Naturally, it is recommended to download the solution only **after** attempting the question.

Disclaimers

Before starting, download the file below and run it to make sure all relevant libraries are imported (the content of the file will be explained at the beginning of the presentation).

`https://www.benoitcorsini.com/files/matplotlib/q0.py`

- This presentation is composed of several coding and visual questions.
  - A "correct" answer is not a word for word file or a pixel for pixel image, but rather a method that provides a similar outcome.
  - The solution of the different questions are available online, by replacing the 0 from the above link with the corresponding question number.
  - Naturally, it is recommended to download the solution only **after** attempting the question.
- `matplotlib` is a massive package and not all methods and attributes can be explained in detail here: finding the relevant information is also part of this workshop.

Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

Importing packages

Importing packages

Before starting, we need to import a few packages.

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.
- `os`: a package for handling files.

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.
- `os`: a package for handling files.
- `cv2`: a package for creating videos.

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.
- `os`: a package for handling files.
- `cv2`: a package for creating videos.
- `matplotlib`: a package for creating images and figures, with many subclasses. Among the relevant subclasses, the presentation focuses on

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.
- `os`: a package for handling files.
- `cv2`: a package for creating videos.
- `matplotlib`: a package for creating images and figures, with many subclasses. Among the relevant subclasses, the presentation focuses on
  - `.patches`: a set of classes to create shapes on the figure.

Importing packages

Before starting, we need to import a few packages.

- `numpy`: a package for handling arrays.
- `os`: a package for handling files.
- `cv2`: a package for creating videos.
- `matplotlib`: a package for creating images and figures, with many subclasses. Among the relevant subclasses, the presentation focuses on
  - `.patches`: a set of classes to create shapes on the figure.
  - `.path`: a set of classes to handle lines and drawings.

Importing packages

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import *
from matplotlib.path import Path
from matplotlib.figure import Figure
from matplotlib.text import TextPath
```

Initializing the file

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).
- We create a new element of the `Visual` class by calling `vis = Visual()`.

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).
- We create a new element of the `Visual` class by calling `vis = Visual()`.
- We run built-in methods by calling `vis.some_method(...)`.

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).
- We create a new element of the `Visual` class by calling `vis = Visual()`.
- We run built-in methods by calling `vis.some_method(...)`.

For the sake of this presentation, our class has two particularities.

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).
- We create a new element of the `Visual` class by calling `vis = Visual()`.
- We run built-in methods by calling `vis.some_method(...)`.

For the sake of this presentation, our class has two particularities.

- It is a subclass of the `Figure` class from `matplotlib` and thus uses pre-implemented arguments and methods using `*args`, `**kwargs`, and `super()`.

Initializing the file

We now create a new class called `Visual`, which we use to organize our code.

- Classes are objects with a set of built-in methods (for example `float`, `list`, `dict`, etc).
- We create a new element of the `Visual` class by calling `vis = Visual()`.
- We run built-in methods by calling `vis.some_method(...)`.

For the sake of this presentation, our class has two particularities.

- It is a subclass of the `Figure` class from `matplotlib` and thus uses pre-implemented arguments and methods using `*args`, `**kwargs`, and `super()`.
- It has a default `square` method using the `@classmethod` decorator, which creates a square figure and obtained by calling `vis = Visual.square(...)`.

Initializing the file

```
class Visual(Figure):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
```

Initializing the file

```
class Visual(Figure):  ..... creates a subclass of Figure

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
```

Initializing the file

```
class Visual(Figure):  ..... creates a subclass of Figure

    def __init__(self, *args, **kwargs):  ..... allows pre-existing arguments
        super().__init__(*args, **kwargs)

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
```

Initializing the file

```
class Visual(Figure):                                   creates a subclass of Figure

    def __init__(self, *args, **kwargs):                allows pre-existing arguments
        super().__init__(*args, **kwargs)               applies the arguments to the superclass

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
```

Initializing the file

```
class Visual(Figure): ..... creates a subclass of Figure

    def __init__(self, *args, **kwargs): ..... allows pre-existing arguments
        super().__init__(*args, **kwargs) ..... applies the arguments to the superclass

    @classmethod ..... creates a default squared structure
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
```

Initializing the file

```
class Visual(Figure): ..... creates a subclass of Figure

    def __init__(self, *args, **kwargs): ..... allows pre-existing arguments
        super().__init__(*args, **kwargs) ..... applies the arguments to the superclass

    @classmethod ..... creates a default squared structure
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)



if __name__ == '__main__': ..... prevents this code from running elsewhere
    vis = Visual.square(dpi=500)
```

Initializing the figure and axes

Initializing the figure and axes

Since we want to create visuals and not graphs, we need to remove the extra space and axes that are part of a default `Figure` element.

Initializing the figure and axes

Since we want to create visuals and not graphs, we need to remove the extra space and axes that are part of a default `Figure` element.

- In `matplotlib`, the `Figure` class always requires at least one `Axes` class in order to represent objects on the figure, using the `add_subplot(...)` method.

Initializing the figure and axes

Since we want to create visuals and not graphs, we need to remove the extra space and axes that are part of a default `Figure` element.

- In `matplotlib`, the `Figure` class always requires at least one `Axes` class in order to represent objects on the figure, using the `add_subplot(...)` method.
- In order to remove the extra space around the figure, we call `subplots_adjust(...)`.

Initializing the figure and axes

Since we want to create visuals and not graphs, we need to remove the extra space and axes that are part of a default `Figure` element.

- In `matplotlib`, the `Figure` class always requires at least one `Axes` class in order to represent objects on the figure, using the `add_subplot(...)` method.
- In order to remove the extra space around the figure, we call `subplots_adjust(...)`.
- In order to hide the axes on the figure, we call `set_axis_off(...)`

Initializing the figure and axes

Since we want to create visuals and not graphs, we need to remove the extra space and axes that are part of a default `Figure` element.

- In `matplotlib`, the `Figure` class always requires at least one `Axes` class in order to represent objects on the figure, using the `add_subplot(...)` method.
- In order to remove the extra space around the figure, we call `subplots_adjust(...)`.
- In order to hide the axes on the figure, we call `set_axis_off(...)`
- Finally, we put this code into a hidden *dunder* (double-under) method for clarity.

Initializing the figure and axes

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__()

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)

    def __figure__(self):
        self.subplots_adjust(left=0, right=1, bottom=0, top=1)
        self.ax = self.add_subplot()
        self.ax.set_axis_off()
```

Initializing the figure and axes

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__() ..... a hidden dunder method

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)

    def __figure__(self):
        self.subplots_adjust(left=0, right=1, bottom=0, top=1)
        self.ax = self.add_subplot()
        self.ax.set_axis_off()
```

Initializing the figure and axes

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__() ..... a hidden dunder method

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)

    def __figure__(self):
        self.subplots_adjust(left=0, right=1, bottom=0, top=1) ..... removes the extra space around the figure
        self.ax = self.add_subplot()
        self.ax.set_axis_off()
```

Initializing the figure and axes

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__() ..... a hidden dunder method

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)

    def __figure__(self):
        self.subplots_adjust(left=0, right=1, bottom=0, top=1) ..... removes the extra space around the figure
        self.ax = self.add_subplot() ..... adds a new Axes element
        self.ax.set_axis_off()
```

Initializing the figure and axes

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__() ..... a hidden dunder method

    @classmethod
    def square(cls, *args, **kwargs):
        return cls(figsize=(1, 1), *args, **kwargs)

    def __figure__(self):
        self.subplots_adjust(left=0, right=1, bottom=0, top=1) ..... removes the extra space around the figure
        self.ax = self.add_subplot() ..... adds a new Axes element
        self.ax.set_axis_off() ..... removes the axis from the figure
```

Auto-creating frames

Auto-creating frames

Since we are interested in creating videos, we add a couple of attributes and methods.

Auto-creating frames

Since we are interested in creating videos, we add a couple of attributes and methods.

- For the video renderer, we need to have the number of frames per second (`fps`) which we add as an input of the class.

Auto-creating frames

Since we are interested in creating videos, we add a couple of attributes and methods.

- For the video renderer, we need to have the number of frames per second (`fps`) which we add as an input of the class.
- To simplify the creation of the different frames, we implement a `new_frame` method, which automatically organizes saves the current state of the figure and increases the frame count by 1.

Auto-creating frames

```
14.     def __init__(self, fps=30, *args, **kwargs):
15.         super().__init__(*args, **kwargs)
16.         self.__figure__()
17.         self.fps = fps
18.         self.frame_index = 0

29.     def new_frame(self):
30.         if not os.path.exists('frames'):
31.             os.mkdir('frames')
32.         self.savefig(f'frames/{self.frame_index:04d}.png')
33.         self.frame_index += 1
```

Auto-creating frames

```
14.     def __init__(self, fps=30, *args, **kwargs):
15.         super().__init__(*args, **kwargs)
16.         self.__figure__()
17.         self.fps = fps ..... adds the frames per second parameter
18.         self.frame_index = 0

29.     def new_frame(self):
30.         if not os.path.exists('frames'):
31.             os.mkdir('frames')
32.         self.savefig(f'frames/{self.frame_index:04d}.png')
33.         self.frame_index += 1
```

Auto-creating frames

```
14.     def __init__(self, fps=30, *args, **kwargs):
15.         super().__init__(*args, **kwargs)
16.         self.__figure__()
17.         self.fps = fps ..... adds the frames per second parameter
18.         self.frame_index = 0 ..... an attribute to index the frames

29.     def new_frame(self):
30.         if not os.path.exists('frames'):
31.             os.mkdir('frames')
32.         self.savefig(f'frames/{self.frame_index:04d}.png')
33.         self.frame_index += 1
```

Auto-creating frames

```
    def __init__(self, fps=30, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__()
        self.fps = fps ..... adds the frames per second parameter
        self.frame_index = 0 ..... an attribute to index the frames

    def new_frame(self):
        if not os.path.exists('frames'):
            os.mkdir('frames') ..... creates a folder for the frames
        self.savefig(f'frames/{self.frame_index:04d}.png')
        self.frame_index += 1
```

Auto-creating frames

```
    def __init__(self, fps=30, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__figure__()
        self.fps = fps ..... adds the frames per second parameter
        self.frame_index = 0 ..... an attribute to index the frames

    def new_frame(self):
        if not os.path.exists('frames'):
            os.mkdir('frames') ..... creates a folder for the frames
        self.savefig(f'frames/{self.frame_index:04d}.png') ..... saves the current state of the figure
        self.frame_index += 1
```

Combining frames for the video

Combining frames for the video

Finally, we need to combine all the created frames into a single video using the `cv2` package.

Combining frames for the video

Finally, we need to combine all the created frames into a single video using the `cv2` package.

→ The structure `make_video` strongly relies on the structure of the `cv2` package, in particular the existence of *silent errors*, halting the code prior to completion without raising any error.

Combining frames for the video

```
    def make_video(self, filename='video'):
        frames = sorted([
            os.path.join('frames', file)
            for file in os.listdir('frames')
        ])
        height, width, _ = cv2.imread(frames[0]).shape
        video = cv2.VideoWriter(
            filename=filename + '.mp4',
            fourcc=cv2.VideoWriter_fourcc(*'mp4v'),
            fps=self.fps,
            frameSize=(width, height),
        )
        for frame in frames:
            video.write(cv2.imread(frame))
        video.release()
        cv2.destroyAllWindows()
```

Combining frames for the video

```
35.     def make_video(self, filename='video'):
36.         frames = sorted([
37.             os.path.join('frames', file)
38.             for file in os.listdir('frames')
39.         ]) ..... lists the frame files in order of creation
40.         height, width, _ = cv2.imread(frames[0]).shape
41.         video = cv2.VideoWriter(
42.             filename=filename + '.mp4',
43.             fourcc=cv2.VideoWriter_fourcc(*'mp4v'),
44.             fps=self.fps,
45.             frameSize=(width, height),
46.         )
47.         for frame in frames:
48.             video.write(cv2.imread(frame))
49.         video.release()
50.         cv2.destroyAllWindows()
```

Combining frames for the video

```
35.     def make_video(self, filename='video'):
36.         frames = sorted([
37.             os.path.join('frames', file)
38.             for file in os.listdir('frames')
39.         ]) ..... lists the frame files in order of creation
40.         height, width, _ = cv2.imread(frames[0]).shape ..... finds the size of the images (to avoid silent errors)
41.         video = cv2.VideoWriter(
42.             filename=filename + '.mp4',
43.             fourcc=cv2.VideoWriter_fourcc(*'mp4v'),
44.             fps=self.fps,
45.             frameSize=(width, height),
46.         )
47.         for frame in frames:
48.             video.write(cv2.imread(frame))
49.         video.release()
50.         cv2.destroyAllWindows()
```

Combining frames for the video

```
35.     def make_video(self, filename='video'):
36.         frames = sorted([
37.             os.path.join('frames', file)
38.             for file in os.listdir('frames')
39.         ]) ..... lists the frame files in order of creation
40.         height, width, _ = cv2.imread(frames[0]).shape ..... finds the size of the images (to avoid silent errors)
41.         video = cv2.VideoWriter(
42.             filename=filename + '.mp4',
43.             fourcc=cv2.VideoWriter_fourcc(*'mp4v'), ..... the format of the video (mp4 here)
44.             fps=self.fps,
45.             frameSize=(width, height),
46.         )
47.         for frame in frames:
48.             video.write(cv2.imread(frame))
49.         video.release()
50.         cv2.destroyAllWindows()
```

Combining frames for the video

```
    def make_video(self, filename='video'):
        frames = sorted([
            os.path.join('frames', file)
            for file in os.listdir('frames')
        ]) ..... lists the frame files in order of creation
        height, width, _ = cv2.imread(frames[0]).shape ..... finds the size of the images (to avoid silent errors)
        video = cv2.VideoWriter(
            filename=filename + '.mp4',
            fourcc=cv2.VideoWriter_fourcc(*'mp4v'), ..... the format of the video (mp4 here)
            fps=self.fps,
            frameSize=(width, height),
        )
        for frame in frames:
            video.write(cv2.imread(frame)) ..... adds each individual frame to the video
        video.release()
        cv2.destroyAllWindows()
```

Combining frames for the video

```
35.     def make_video(self, filename='video'):
36.         frames = sorted([
37.             os.path.join('frames', file)
38.             for file in os.listdir('frames')
39.         ]) ..... lists the frame files in order of creation
40.         height, width, _ = cv2.imread(frames[0]).shape ..... finds the size of the images (to avoid silent errors)
41.         video = cv2.VideoWriter(
42.             filename=filename + '.mp4',
43.             fourcc=cv2.VideoWriter_fourcc(*'mp4v'), ..... the format of the video (mp4 here)
44.             fps=self.fps,
45.             frameSize=(width, height),
46.         )
47.         for frame in frames:
48.             video.write(cv2.imread(frame)) ..... adds each individual frame to the video
49.         video.release() ..... saves the video
50.         cv2.destroyAllWindows()
```

Combining frames for the video

```
    def make_video(self, filename='video'):
        frames = sorted([
            os.path.join('frames', file)
            for file in os.listdir('frames')
        ]) ..... lists the frame files in order of creation
        height, width, _ = cv2.imread(frames[0]).shape ..... finds the size of the images (to avoid silent errors)
        video = cv2.VideoWriter(
            filename=filename + '.mp4',
            fourcc=cv2.VideoWriter_fourcc(*'mp4v'), ..... the format of the video (mp4 here)
            fps=self.fps,
            frameSize=(width, height),
        )
        for frame in frames:
            video.write(cv2.imread(frame)) ..... adds each individual frame to the video
        video.release() ..... saves the video
        cv2.destroyAllWindows() ..... clears the cache
```

A first video

A first video

Q1: Create a 5 seconds video using the code above, also available at:

https://www.benoitcorsini.com/files/matplotlib/q0.py

A first video

```
    def duration_to_number(self, duration):
        return int(duration*self.fps)

    def wait(self, duration):
        for _ in range(self.duration_to_number(duration)):
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.wait(5)
    vis.make_video()
```

A first video

```
    def duration_to_number(self, duration):  ..... transforms a duration (in seconds) into a number of frames
        return int(duration*self.fps)

    def wait(self, duration):
        for _ in range(self.duration_to_number(duration)):
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.wait(5)
    vis.make_video()
```

A first video

```
    def duration_to_number(self, duration): ..... transforms a duration (in seconds) into a number of frames
        return int(duration*self.fps)

    def wait(self, duration): ..... pauses the current state of the figure for a given duration
        for _ in range(self.duration_to_number(duration)):
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.wait(5)
    vis.make_video()
```

Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

Adding an image

Adding an image

A few useful functions to import and manipulate images.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.
  - For example, we make it invisible by calling `image.set(visible=False)`.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.
  - For example, we make it invisible by calling `image.set(visible=False)`.
  - Alternatively, this can also be done by calling `image.set_visible(False)`.

Adding an image

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.
  - For example, we make it invisible by calling `image.set(visible=False)`.
  - Alternatively, this can also be done by calling `image.set_visible(False)`.

Q2: Import an image into the figure.

Adding an image

```
    def set_boundary(self, boundary=1):
        self.ax.set_xlim(-boundary, boundary)
        self.ax.set_ylim(-boundary, boundary)

    def add_image(self, filename, shift=0, *args, **kwargs):
        X = plt.imread(filename)
        self.image = self.ax.imshow(
            X=X,
            extent=(
                shift - 1 - 2*X.shape[0]/X.shape[1],
                shift + 1,
                -1,
                1,
            ),
            *args,
            **kwargs,
        )
```

Adding an image

```
if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.add_image(filename='singapore.jpg', shift=0.4)
    vis.new_frame()
```

Making the image appear

Making the image appear

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.
  - For example, we make it invisible by calling `image.set(visible=False)`.
  - Alternatively, this can also be done by calling `image.set_visible(False)`.

Making the image appear

A few useful functions to import and manipulate images.

- We transform an image into a matrix by calling `X = plt.imread(filename)`.
- We import this matrix as in image in our figure by calling `image = self.ax.imshow(X, ...)`.
- We can later modify the image by calling `image.set(...)`.
  - For example, we make it invisible by calling `image.set(visible=False)`.
  - Alternatively, this can also be done by calling `image.set_visible(False)`.

Q3: Make an image fade in.

Making the image appear

```
    def image_appear(self, duration):
        n_steps = self.duration_to_number(duration)
        for step in range(n_steps):
            self.image.set_alpha((1 + step)/n_steps)
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.add_image(filename='singapore.jpg', shift=0.4)
    vis.image.set_alpha(0)
    vis.new_frame()
    vis.image_appear(0.1)
```

Making the image appear

```
    def image_appear(self, duration):
        n_steps = self.duration_to_number(duration)
        for step in range(n_steps):
            self.image.set_alpha((1 + step)/n_steps)
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.add_image(filename='singapore.jpg', shift=0.4)
    vis.image.set_alpha(0)
    vis.new_frame()
    vis.image_appear(0.1)
```

Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

Shapes in matplotlib

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the classes, each with its own inputs.

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the classes, each with its own inputs.
- `patch = Circle((0, 0), ...)` creates a circle patch centered at the origin.

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the classes, each with its own inputs.
- `patch = Circle((0, 0), ...)` creates a circle patch centered at the origin.
- `ax.add_patch(patch)` adds the patch to the figure.

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the classes, each with its own inputs.
- `patch = Circle((0, 0), ...)` creates a circle patch centered at the origin.
- `ax.add_patch(patch)` adds the patch to the figure.
- `patch = ax.add_patch(Circle((0, 0), ...))` is the same as the previous two steps.

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matplotlib.patches`:

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the classes, each with its own inputs.
- `patch = Circle((0, 0), ...)` creates a circle patch centered at the origin.
- `ax.add_patch(patch)` adds the patch to the figure.
- `patch = ax.add_patch(Circle((0, 0), ...))` is the same as the previous two steps.
- `patch.set(...)` allows to later modify the patch.

Shapes in matplotlib

Shapes in `matplotlib` are handled by subclasses of `matp`

- `Circle`, `Rectangle`, `Polygon`, etc, are some of the
- `patch = Circle((0, 0), ...)` creates a circle pa
- `ax.add_patch(patch)` adds the patch to the figure
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)` allows to later modify the patch.

Shapes in matplotlib

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Shapes in matplotlib

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q4: Create this image.

Shapes in matplotlib

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q4: Create this image.

→ AI solutions (AI-know, ChatGPT, Gemini)

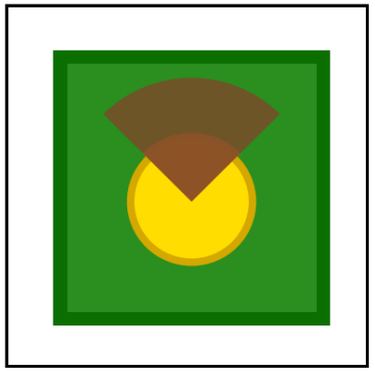
Shapes in matplotlib

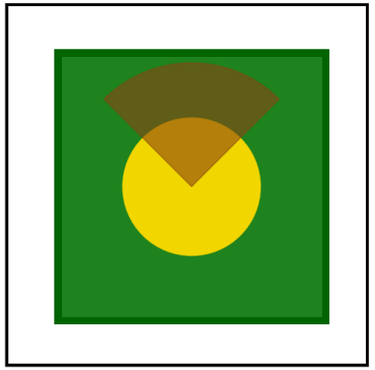
Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q4: Create this image.

→ AI solutions (AI-know, ChatGPT, Gemini)





Shapes in matplotlib

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q4: Create this image.

Shapes in matplotlib

```
if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary(1.1)
    vis.ax.add_patch(Rectangle(
        xy=(-1, -1),
        width=2,
        height=2,
        edgecolor='darkgreen',
        facecolor='forestgreen',
        linewidth=4,
        capstyle='round',
        joinstyle='round',
    ))
```

Shapes in matplotlib

```
    vis.ax.add_patch(Circle(
        xy=(0, 0),
        radius=0.5,
        ec='darkgoldenrod',
        fc='gold',
        lw=2,
    ))
    vis.ax.add_patch(Wedge(
        center=(0, 0),
        theta1=45,
        theta2=135,
        r=1,
        color='crimson',
        lw=0,
        alpha=0.5,
    ))
    vis.new_frame()
```

3D effect

3D effect

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

3D effect

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

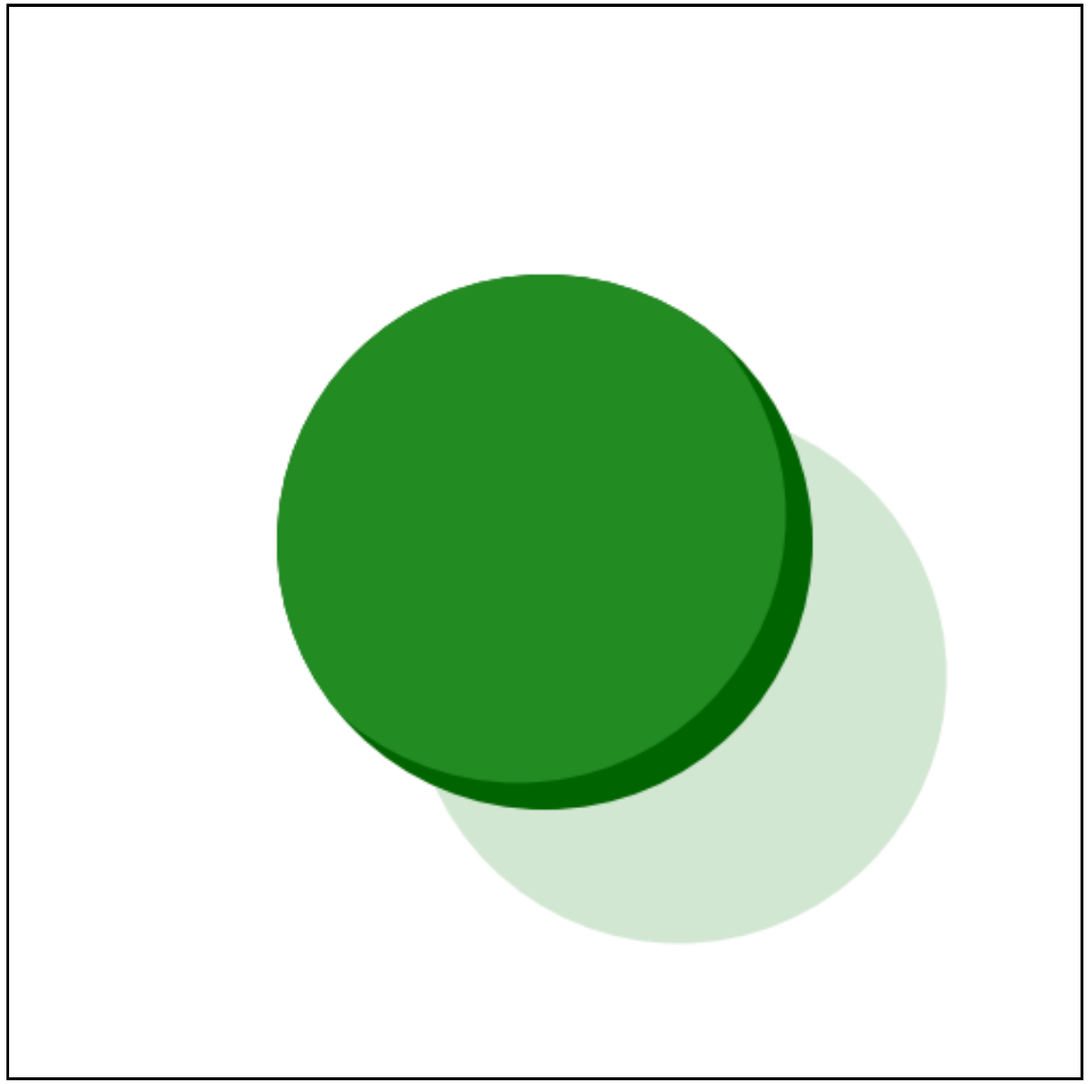
Q5: Create a sphere.

3D effect

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q5: Create a sphere.



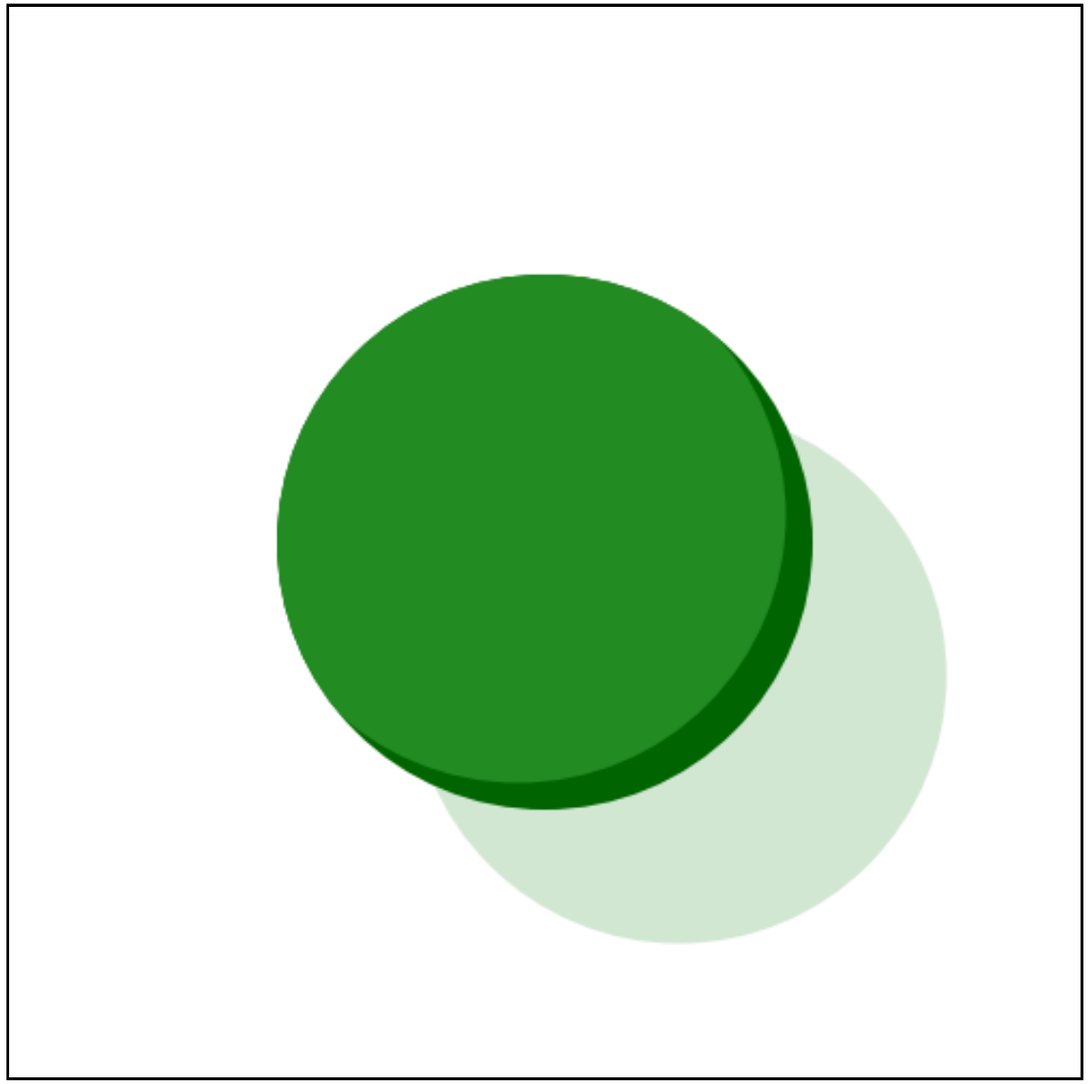
3D effect

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q5: Create a sphere.

→ AI solutions (AI-know, ChatGPT, Gemini)



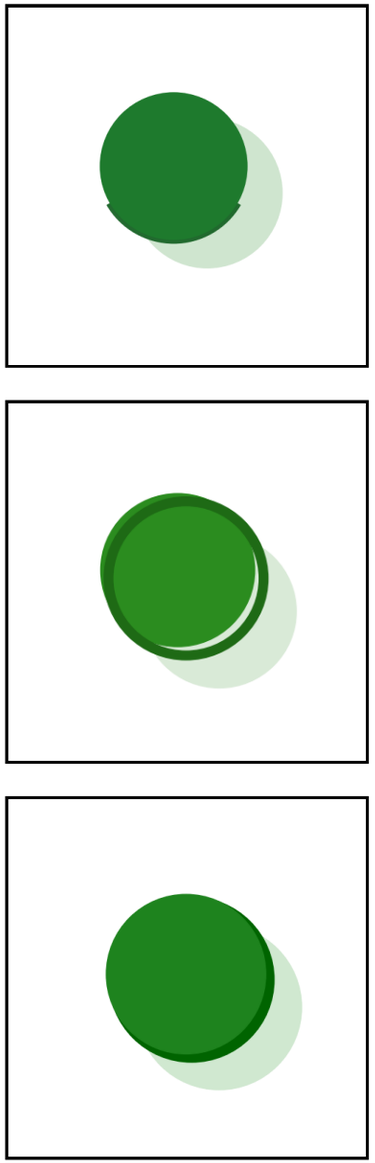
3D effect

Shapes in `matplotlib`:

- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q5: Create a sphere.

→ AI solutions (AI-know, ChatGPT, Gemini)

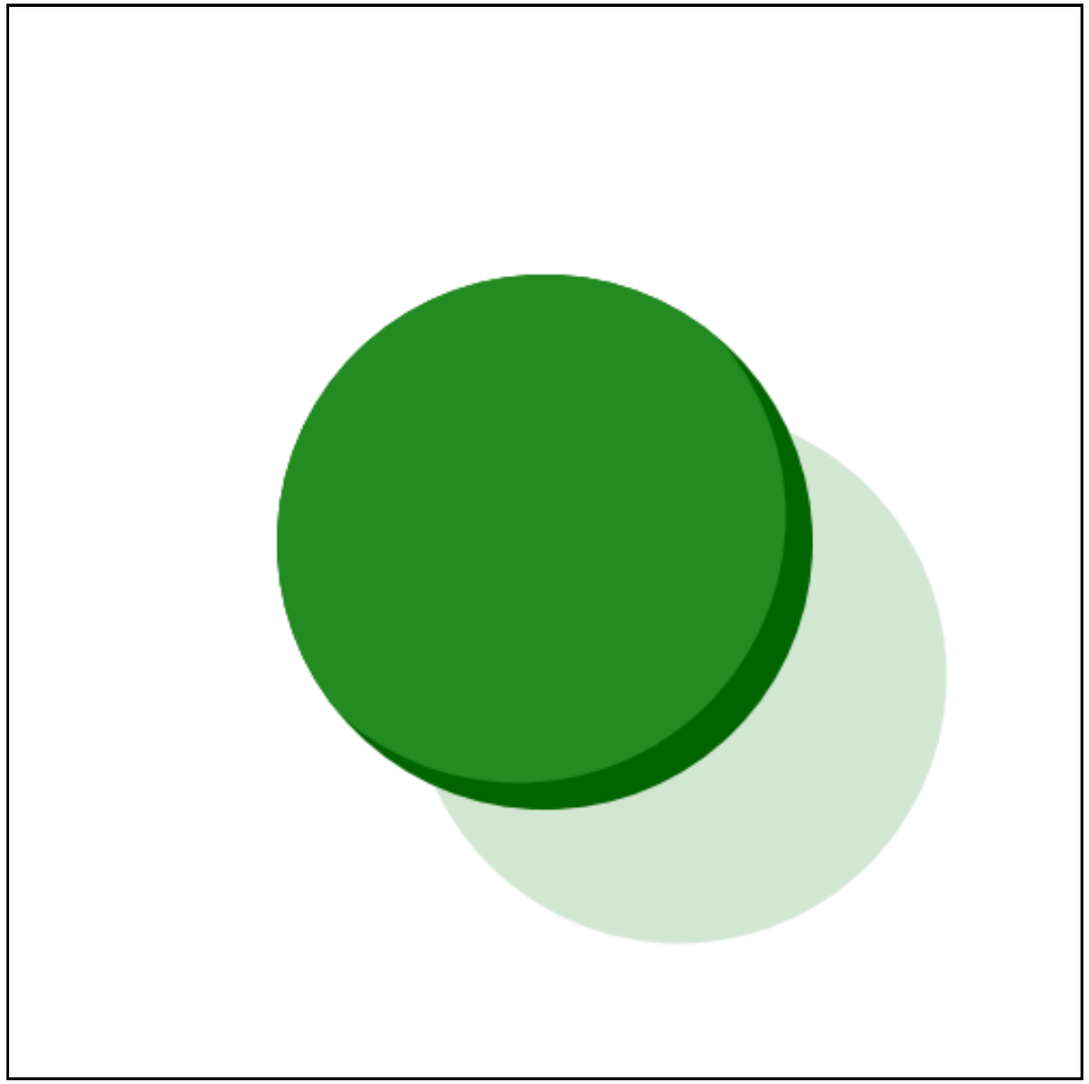


3D effect

Shapes in `matplotlib`:

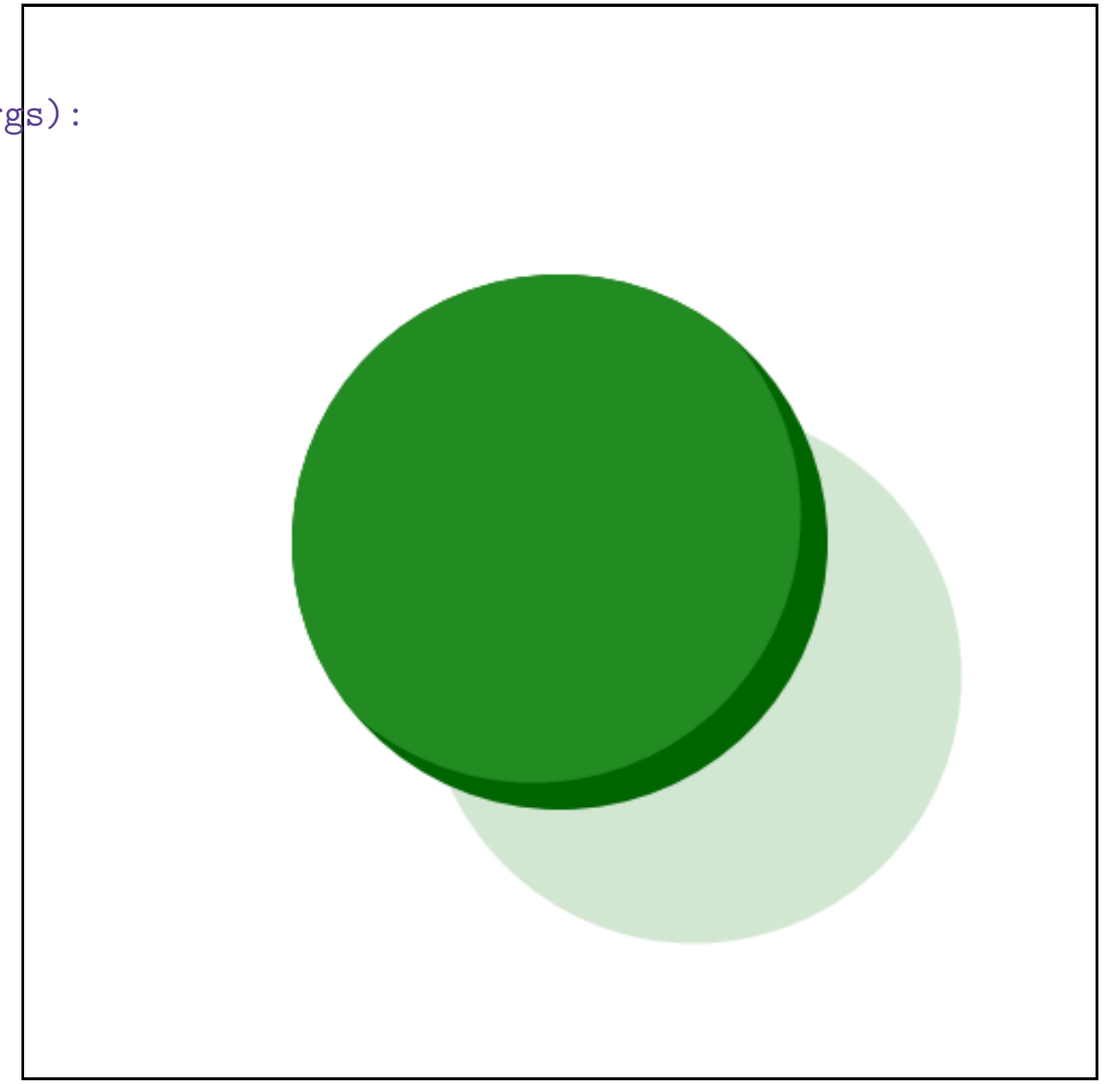
- `Circle`, `Rectangle`, `Polygon`, etc
- `patch = Circle((0, 0), ...)`
- `ax.add_patch(patch)`
- `patch = ax.add_patch(Circle((0, 0), ...))`
- `patch.set(...)`

Q5: Create a sphere.



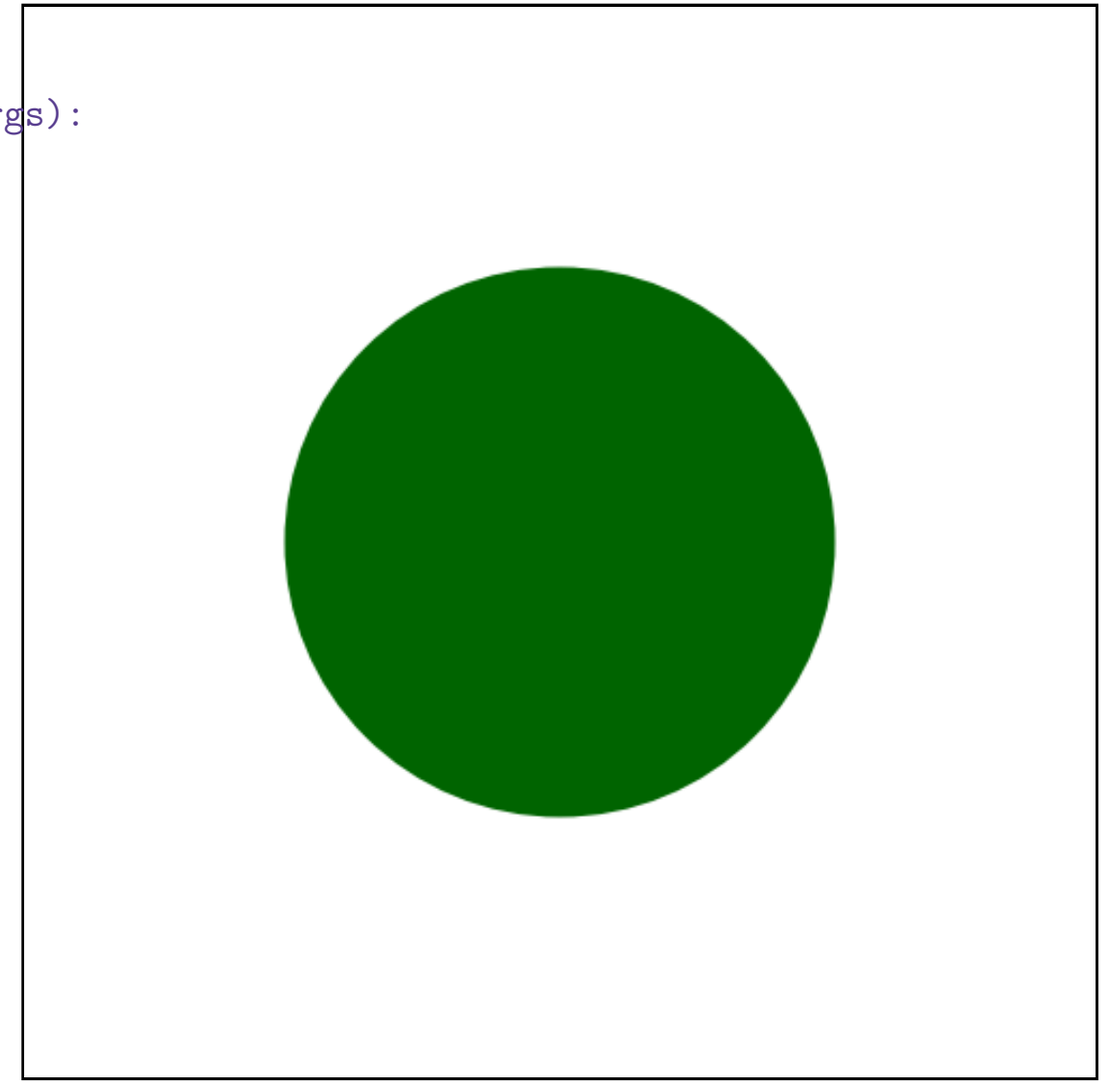
3D effect

```
    def add_circle(self, xy=(0, 0), radius=1, *args, **kwargs):
        return self.ax.add_patch(Circle(
            xy=xy,
            radius=radius,
            *args,
            **kwargs,
        ))



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary(2)
    sphere = vis.add_circle(color='darkgreen')
    vis.new_frame()
```



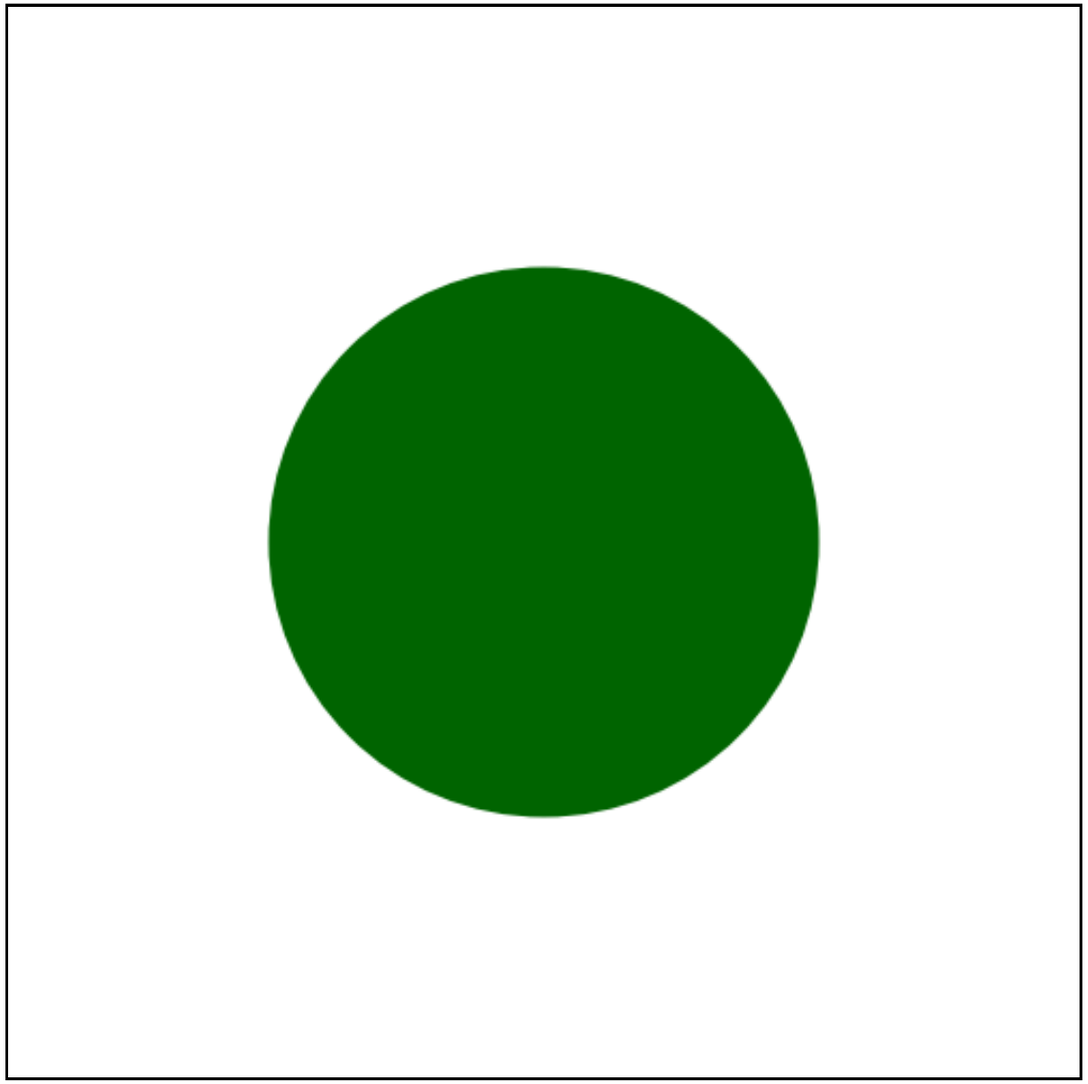
3D effect

```
    def add_circle(self, xy=(0, 0), radius=1, *args, **kwargs):
        return self.ax.add_patch(Circle(
            xy=xy,
            radius=radius,
            *args,
            **kwargs,
        ))



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary(2)
    sphere = vis.add_circle(color='darkgreen')
    vis.new_frame()
```



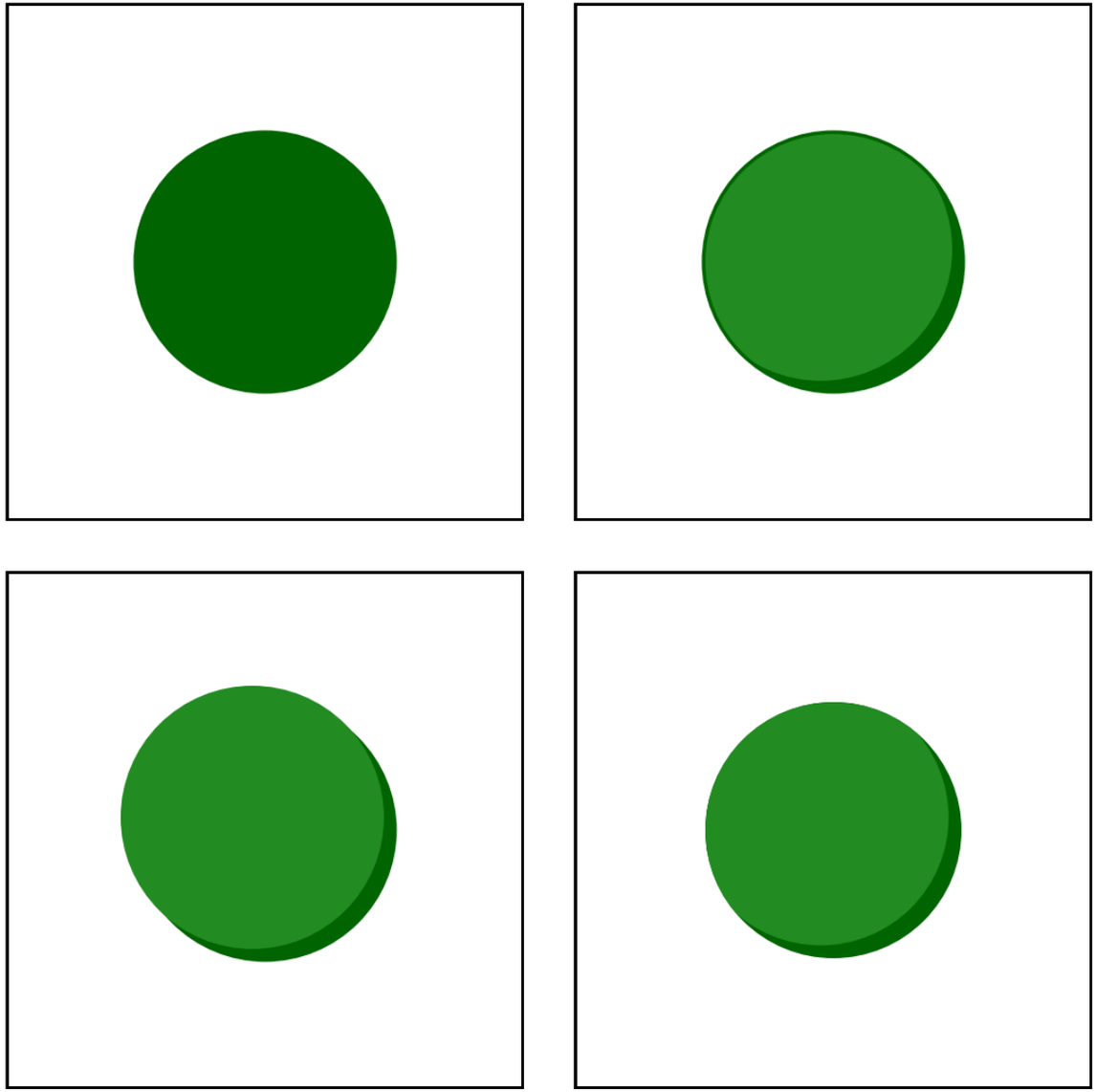
3D effect

```
side_shift = 0.1
light = vis.add_circle(
    xy=(-side_shift, side_shift),
    color='forestgreen',
)
vis.new_frame()
light.set_clip_path(sphere)
vis.new_frame()
sphere.set_lw(0)
light.set_lw(0)
vis.new_frame()
```



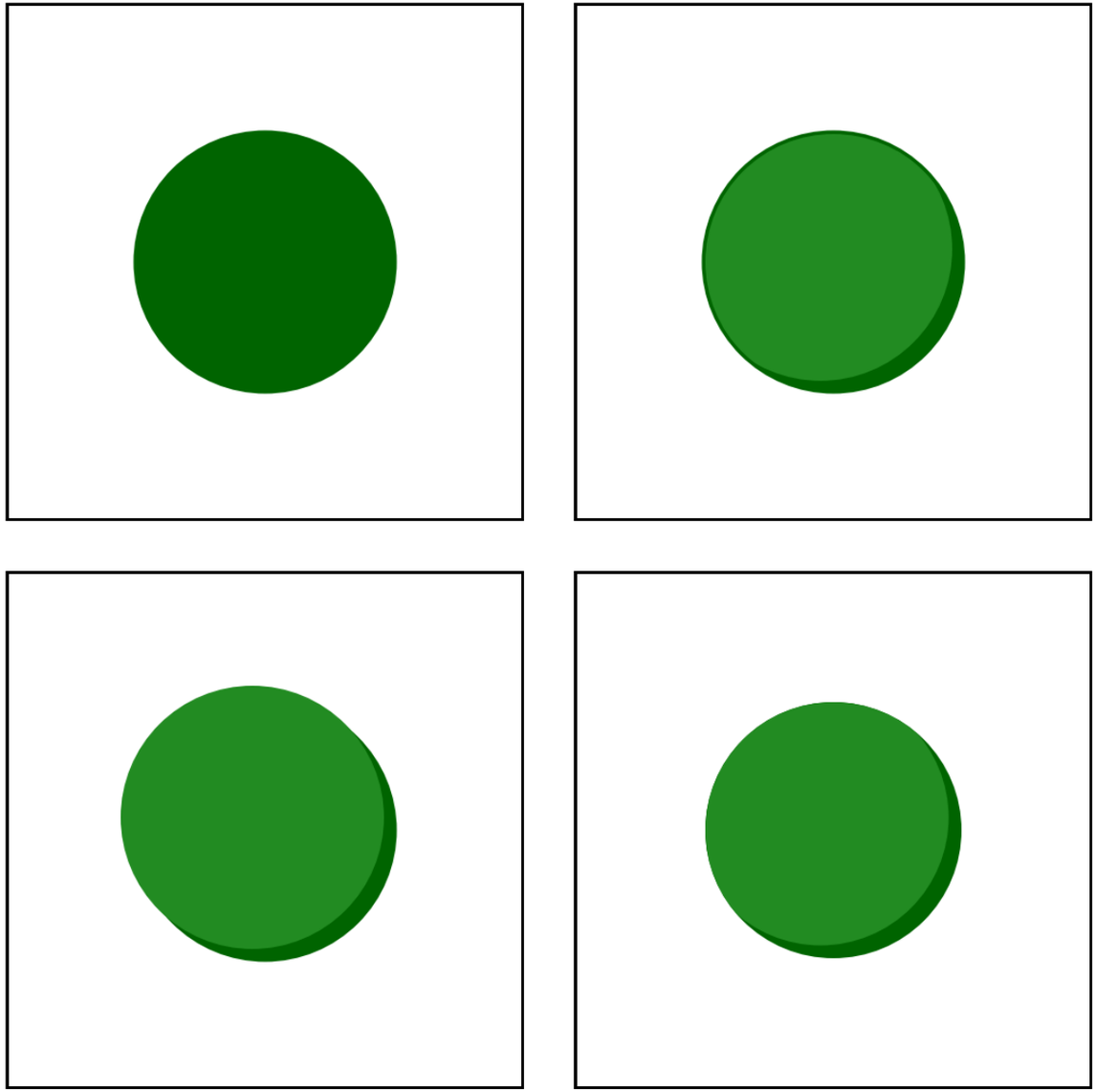
3D effect

```
side_shift = 0.1
light = vis.add_circle(
    xy=(-side_shift, side_shift),
    color='forestgreen',
)
vis.new_frame()
light.set_clip_path(sphere)
vis.new_frame()
sphere.set_lw(0)
light.set_lw(0)
vis.new_frame()
```



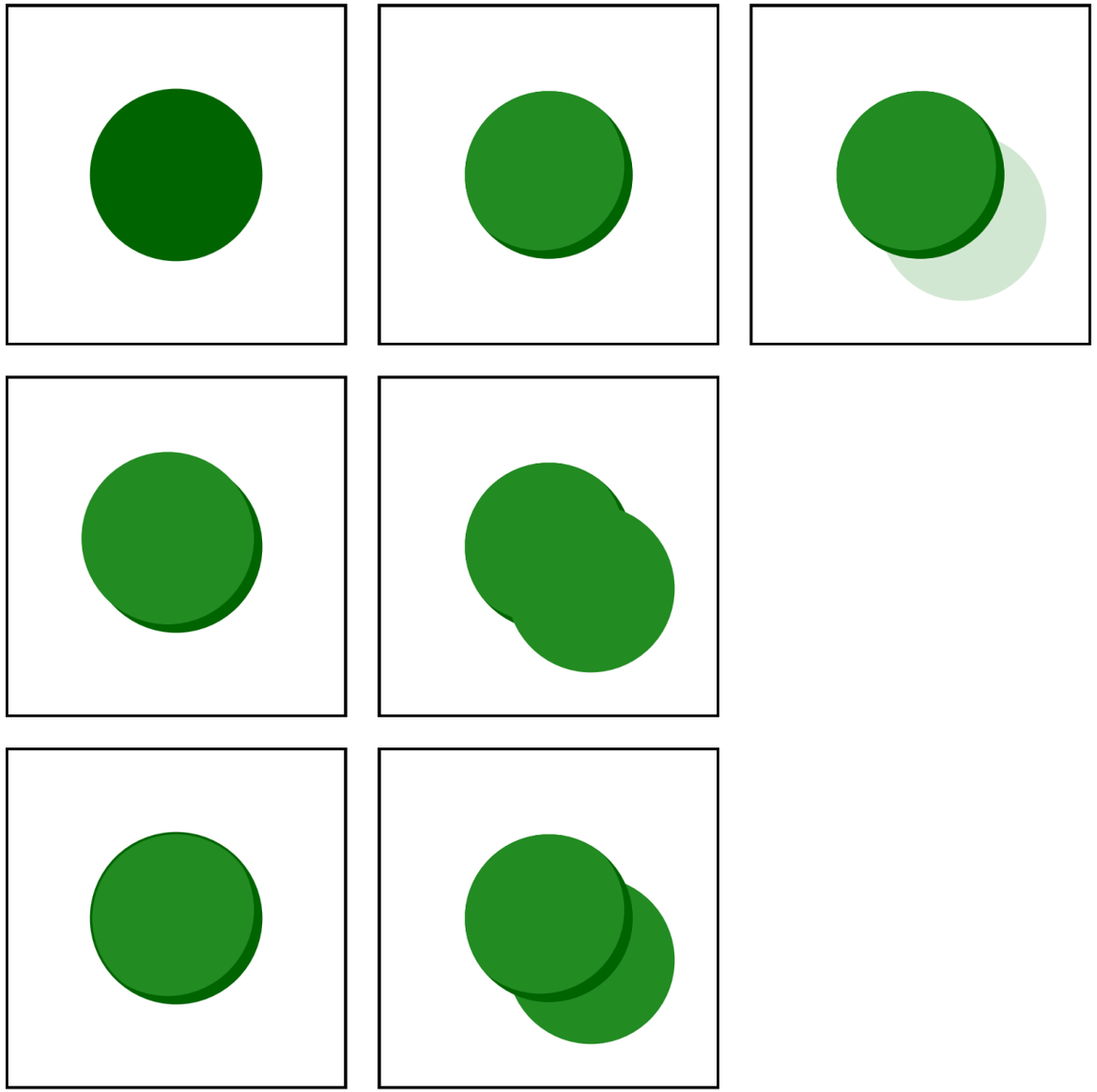
3D effect

```
shade = vis.add_circle(
xy=(0.5, -0.5),
    color='forestgreen',
    lw=0,
)
vis.new_frame()
shade.set_zorder(0)
vis.new_frame()
shade.set_alpha(0.2)
vis.new_frame()
```



3D effect

```
shade = vis.add_circle(
xy=(0.5, -0.5),
    color='forestgreen',
    lw=0,
)
vis.new_frame()
shade.set_zorder(0)
vis.new_frame()
shade.set_alpha(0.2)
vis.new_frame()
```



Creating a sphere

Creating a sphere

Since spheres are created using three circles, it is easier to create methods to handle them together.

Creating a sphere

Since spheres are created using three circles, it is easier to create methods to handle them together.

- We implement `new_sphere`, which creates and returns a dictionary named `sphere`, containing the three circles as well as other parameters (their position, radius, etc).

Creating a sphere

Since spheres are created using three circles, it is easier to create methods to handle them together.

- We implement `new_sphere`, which creates and returns a dictionary named `sphere`, containing the three circles as well as other parameters (their position, radius, etc).
- We implement `update_sphere`, which directly applies to the dictionary of a sphere and should thus be called using `update_sphere(**sphere)`.

Creating a sphere

Since spheres are created using three circles, it is easier to create methods to handle them together.

- We implement `new_sphere`, which creates and returns a dictionary named `sphere`, containing the three circles as well as other parameters (their position, radius, etc).
- We implement `update_sphere`, which directly applies to the dictionary of a sphere and should thus be called using `update_sphere(**sphere)`.

The `new_sphere` and `update_sphere` methods are already in the solution of **Q5**:

https://www.benoitcorsini.com/files/matplotlib/q5.py

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen',
            dark='darkgreen',
            alpha=0.2,
            **kwargs,
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0)
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen',
            alpha=0.2,
            **kwargs,
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0)
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2,
            **kwargs,
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0)
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs,
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0)
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0)
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0) ..... each key leads to a circle patch
            for key in ['main', 'light', 'shade']
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0) ..... each key leads to a circle patch
            for key in ['main', 'light', 'shade'] ..... the sphere is composed of three circles: main, light, and shade
        }
        sphere.update(**kwargs)
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0) ..... each key leads to a circle patch
            for key in ['main', 'light', 'shade'] ..... the sphere is composed of three circles: main, light, and shade
        }
        sphere.update(**kwargs) ..... adds the input of the function as items of the dictionnary sphere
        sphere['main'].set_color(dark)
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0) ..... each key leads to a circle patch
            for key in ['main', 'light', 'shade'] ..... the sphere is composed of three circles: main, light, and shade
        }
        sphere.update(**kwargs) ..... adds the input of the function as items of the dictionnary sphere
        sphere['main'].set_color(dark) ..... the main sphere is the background and darker one
        sphere['shade'].set(alpha=alpha, zorder=0)
        return sphere
```

Creating a sphere

```
    def new_sphere(self,
            color='forestgreen', ..... the front color of the sphere
            dark='darkgreen', ..... the color of the side shade of the sphere
            alpha=0.2, ..... the opacity of the floor shade of the sphere
            **kwargs, ..... other possible arguments of the sphere, as found in update_sphere
        ):
        sphere = {
            key :  vis.add_circle(color=color, lw=0) ..... each key leads to a circle patch
            for key in ['main', 'light', 'shade'] ..... the sphere is composed of three circles: main, light, and shade
        }
        sphere.update(**kwargs) ..... adds the input of the function as items of the dictionnary sphere
        sphere['main'].set_color(dark) ..... the main sphere is the background and darker one
        sphere['shade'].set(alpha=alpha, zorder=0) ..... the shade sphere is the one on the floor
        return sphere
```

Creating a sphere

```
    def update_sphere(self, main, light, shade,
            xy=(0, 0),
            radius=1,
            height=0,
            shift=(0.4, -0.8),
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0),
            radius=1,
            height=0,
            shift=(0.4, -0.8),
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1,
            height=0,
            shift=(0.4, -0.8),
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0,
            shift=(0.4, -0.8),
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8),
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15,
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15, ..... the amount of shift used to create the side shade
            shadow=0.5,
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15, ..... the amount of shift used to create the side shade
            shadow=0.5, ..... the amount of shift used to create the floor shade
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height]))
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15, ..... the amount of shift used to create the side shade
            shadow=0.5, ..... the amount of shift used to create the floor shade
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height])) ..... main is vertically shifted by height from xy
        light.set_center(xy - radius*side*shift + np.array([0, height]))
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15, ..... the amount of shift used to create the side shade
            shadow=0.5, ..... the amount of shift used to create the floor shade
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height])) ..... main is vertically shifted by height from xy
        light.set_center(xy - radius*side*shift + np.array([0, height])) ..... light is shifted from main using shift
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift)
        light.set_clip_path(main)
```

Creating a sphere

```
    def update_sphere(self, main, light, shade, ..... the three circles of the sphere
            xy=(0, 0), ..... the planar position of the sphere
            radius=1, ..... the radius of the sphere
            height=0, ..... the height of the sphere
            shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
            side=0.15, ..... the amount of shift used to create the side shade
            shadow=0.5, ..... the amount of shift used to create the floor shade
        ):
        xy = np.array(xy)
        shift = np.array(shift)
        for circle in [main, light, shade]:
            circle.set_radius(radius)
        main.set_center(xy + np.array([0, height])) ..... main is vertically shifted by height from xy
        light.set_center(xy - radius*side*shift + np.array([0, height])) ..... light is shifted from main using shift
        shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
        shade.set_center(xy + shade_shift*shift) ..... shade is shifted from xy using shift
        light.set_clip_path(main)
```

Creating a sphere

```
106.     def update_sphere(self, main, light, shade, ..... the three circles of the sphere
107.             xy=(0, 0), ..... the planar position of the sphere
108.             radius=1, ..... the radius of the sphere
109.             height=0, ..... the height of the sphere
110.             shift=(0.4, -0.8), ..... the directional shift of the shade of the sphere
111.             side=0.15, ..... the amount of shift used to create the side shade
112.             shadow=0.5, ..... the amount of shift used to create the floor shade
113.         ):
114.         xy = np.array(xy)
115.         shift = np.array(shift)
116.         for circle in [main, light, shade]:
117.             circle.set_radius(radius)
118.         main.set_center(xy + np.array([0, height])) ..... main is vertically shifted by height from xy
119.         light.set_center(xy - radius*side*shift + np.array([0, height])) ..... light is shifted from main using shift
120.         shade_shift = height*shadow + radius/np.sum(shift**2)**0.5
121.         shade.set_center(xy + shade_shift*shift) ..... shade is shifted from xy using shift
122.         light.set_clip_path(main) ..... light is clipped by main
```

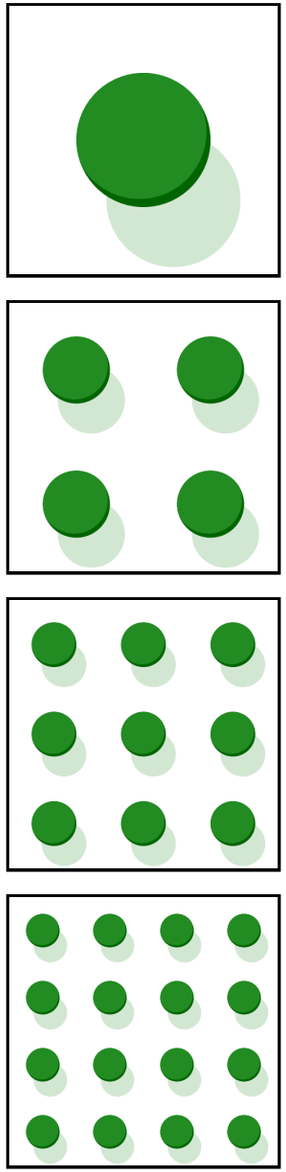
Creating spheres

Creating spheres

Q6: Create a custom number of spheres.

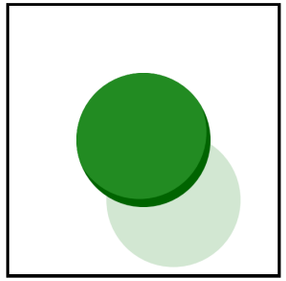
Creating spheres

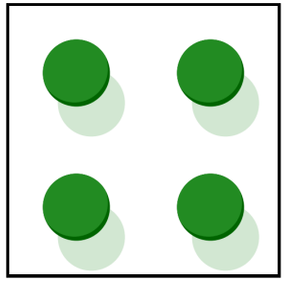
Q6: Create a custom number of spheres.

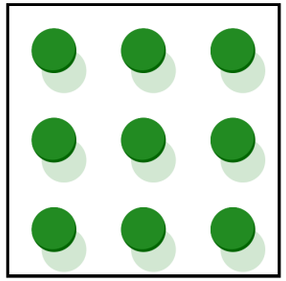


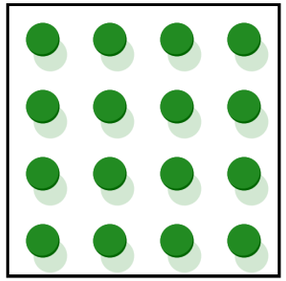
Creating spheres

```
124.     def make_spheres(self, number, ratio=0.5, *args, **kwargs):
125.         self.spheres = []
126.         for i in range(number):
127.             for j in range(number):
128.                 xy = (2*i + 1)/number - 1, 1 - (2*j + 1)/number
129.                 sphere = self.new_sphere(
130.                     radius=ratio/number,
131.                     xy=xy,
132.                     *args,
133.                     **kwargs,
134.                 )
135.                 self.update_sphere(**sphere)
136.                 self.spheres.append(sphere)
```



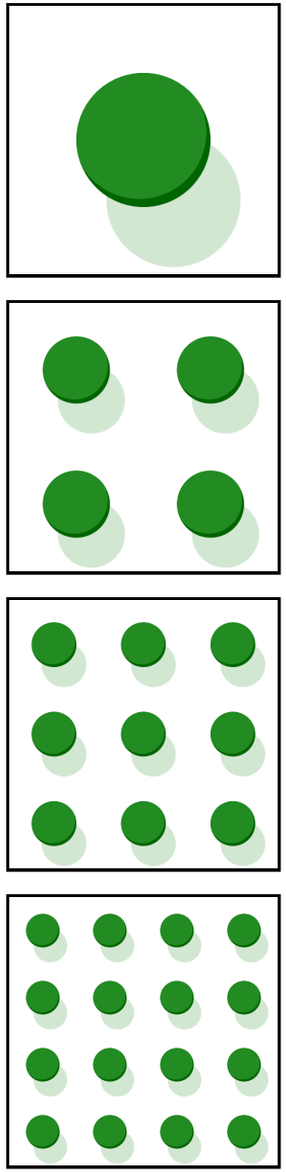






Creating spheres

```
if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    for n in range(1, 5):
        vis.make_spheres(n)
        vis.new_frame()
        for sphere in vis.spheres:
            sphere['main'].set_visible(False)
            sphere['light'].set_visible(False)
            sphere['shade'].set_visible(False)
```



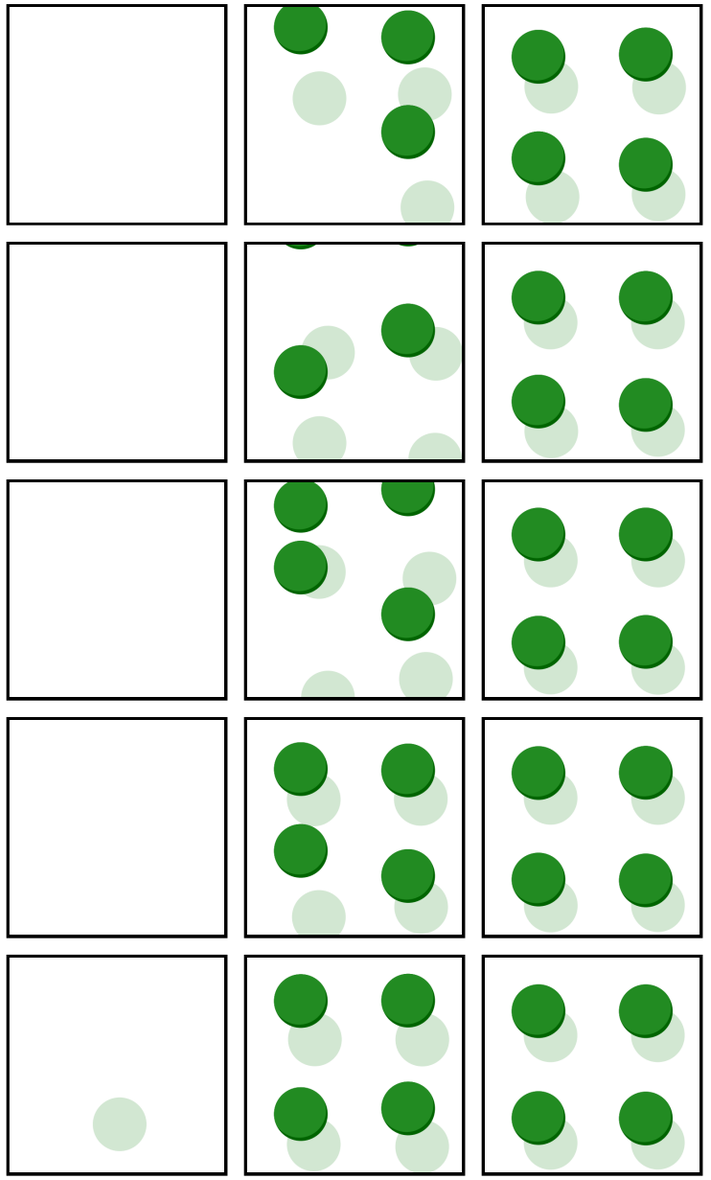
Dropping spheres

Dropping spheres

Q7: Create dropping spheres.

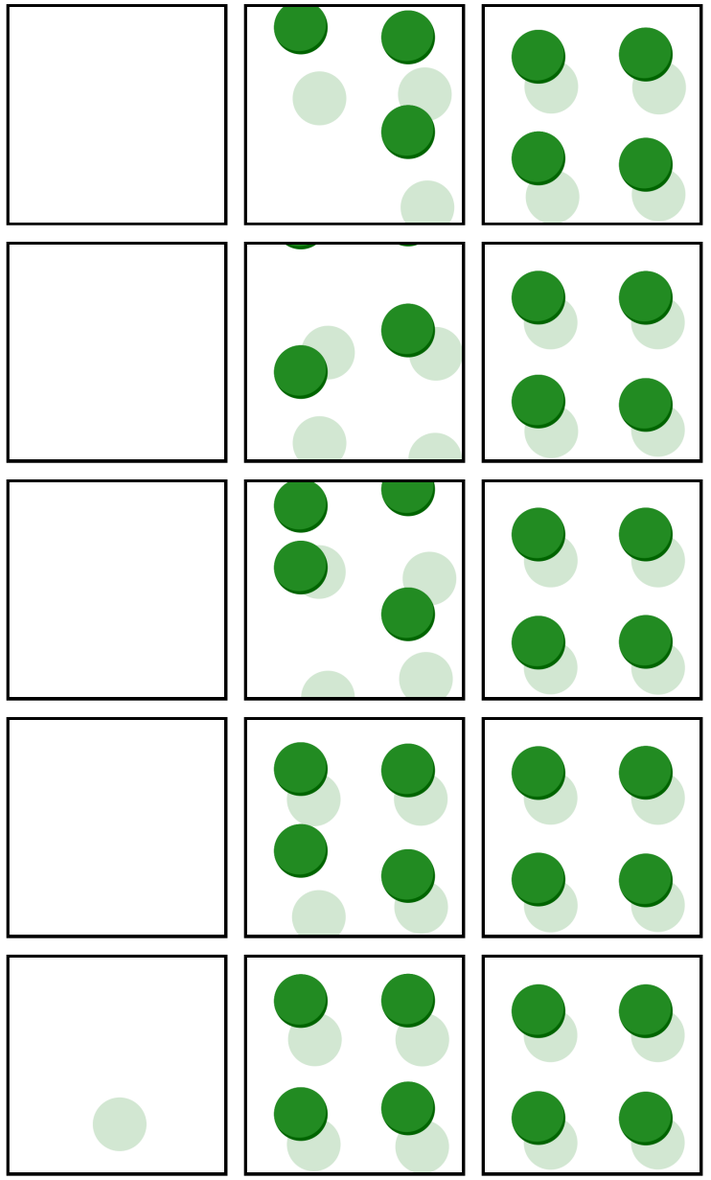
Dropping spheres

Q7: Create dropping spheres.



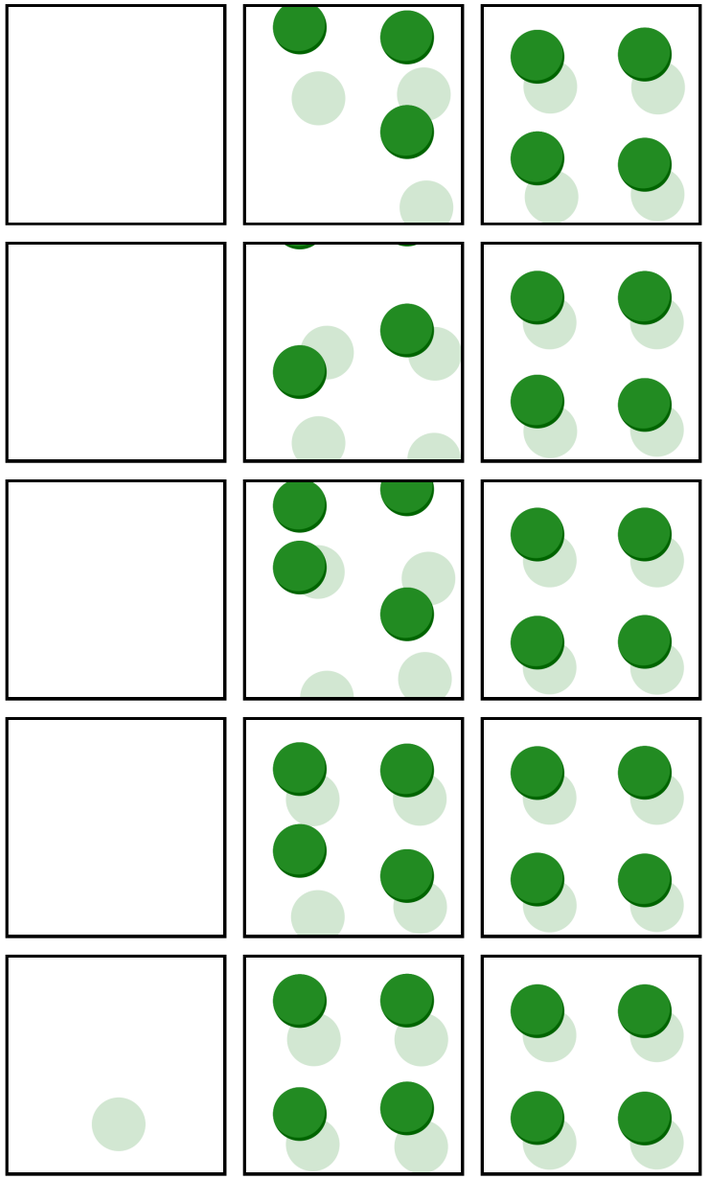
Dropping spheres

```
    def get_height(self,
            index=0,
            height_spread=1,
            height_period=0.5,
            height_drop=0.25,
            max_height=4.5,
        ):
        if not hasattr(self, 'height_shifts'):
            self.height_shifts = np.random.rand(len(self.spheres))
        shifts = index - self.fps*height_spread*self.height_shifts
        shifts = shifts*(shifts >= 0)
        flucts = np.abs(np.cos(np.pi*shifts/self.fps/height_period))
        heights = np.exp(-shifts/self.fps/height_drop)
        return max_height*flucts*heights
```



Dropping spheres

```
    def drop_spheres(self, duration, *args, **kwargs):
        for index in range(self.duration_to_number(duration)):
            heights = self.get_height(index, *args, **kwargs)
            for sphere, height in zip(self.spheres, heights):
                self.update_sphere(height=height, **sphere)
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.make_spheres(2)
    vis.drop_spheres(0.5,
        height_spread=0.2,
        height_period=0.1,
        height_drop=0.05,
    )
```



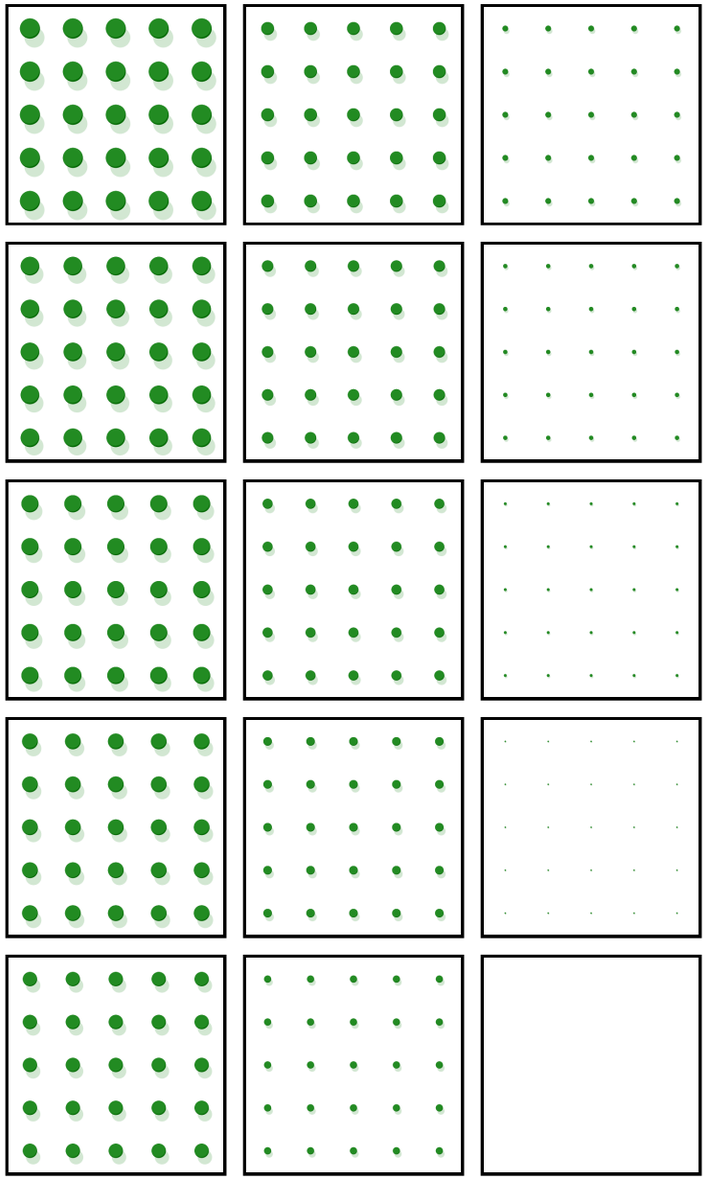
Schrinking spheres

Schrinking spheres

Q8: Create schrinking spheres.

Schrinking spheres

Q8: Create schrinking spheres.



Schrinking spheres

```
    def schrink_spheres(self, duration):
        n_steps = self.duration_to_number(duration)
        radius = self.spheres[0]['main'].get_radius()
        for index in range(n_steps):
            ratio = 1 - (1 + index)/n_steps
            for sphere in self.spheres:
                sphere['radius'] = ratio*radius
                self.update_sphere(**sphere)
            self.new_frame()


if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.make_spheres(5)
    vis.schrink_spheres(0.5)
```

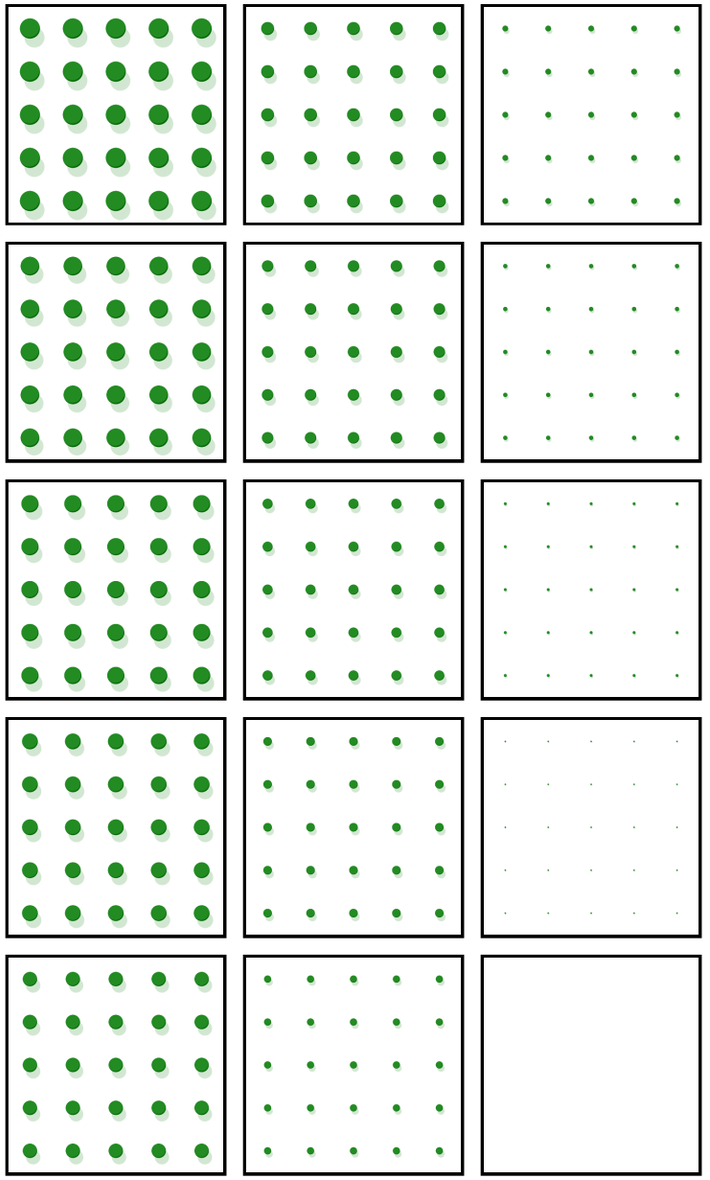


Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

Drawings

Drawings

The `Path` class is useful in creating curved shapes.

Drawings

The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.

Drawings

The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.

Drawings

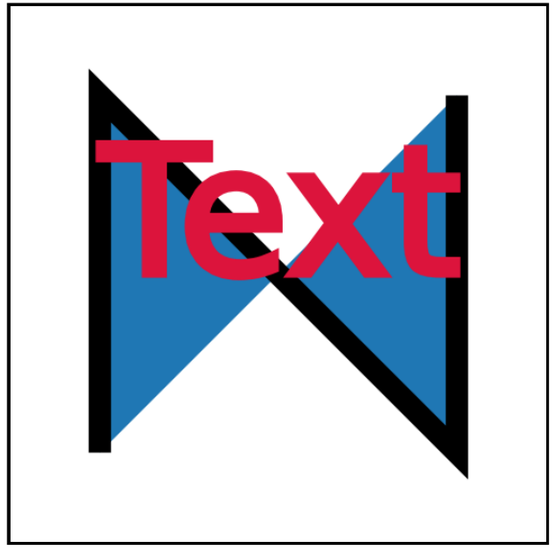
The `Path` class is useful in creating curved shapes.

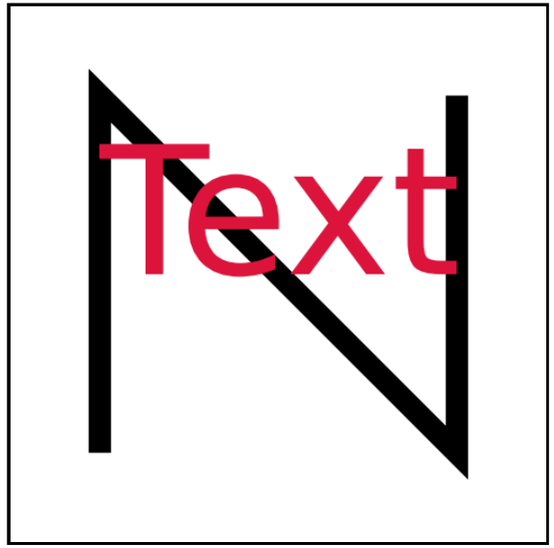
- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.

Drawings

The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.



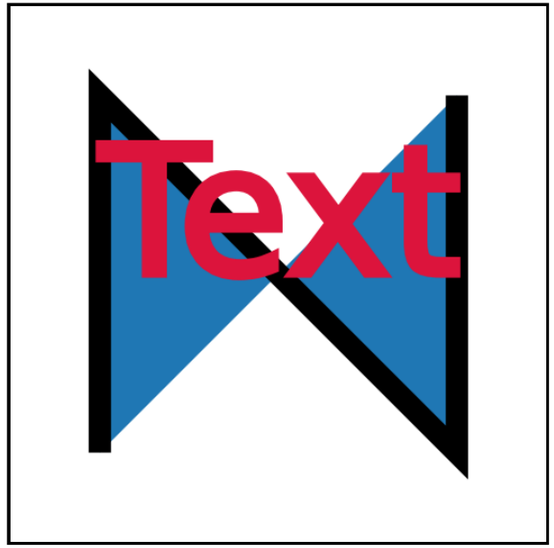


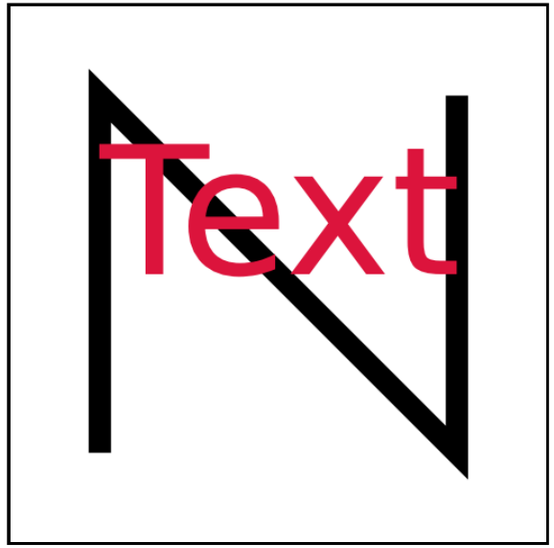
Drawings

The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.

Q9: Create these two images.





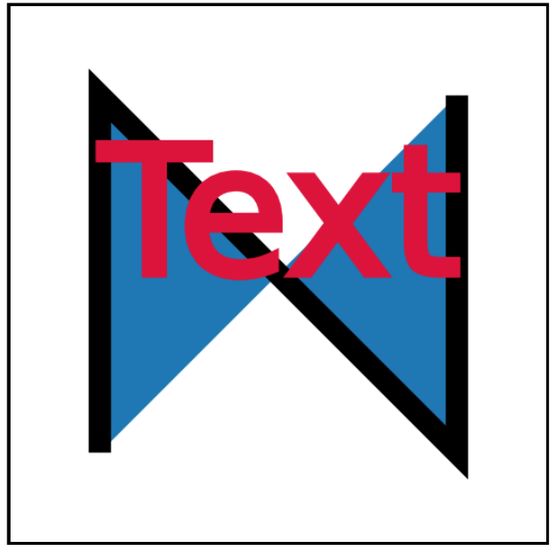
Drawings

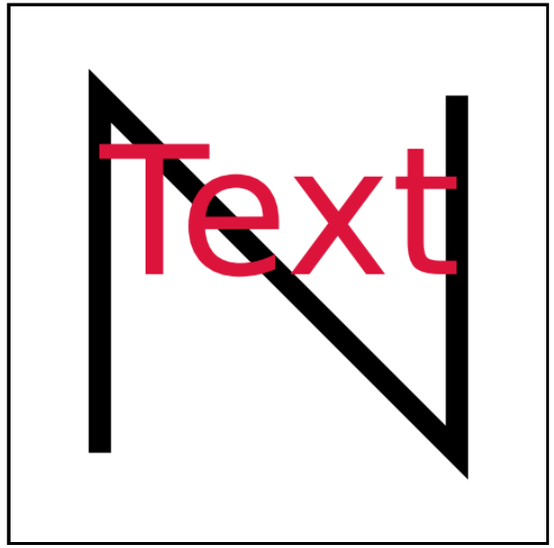
The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.

Q9: Create these two images.

→ AI solutions (AI-know, ChatGPT, Gemini)





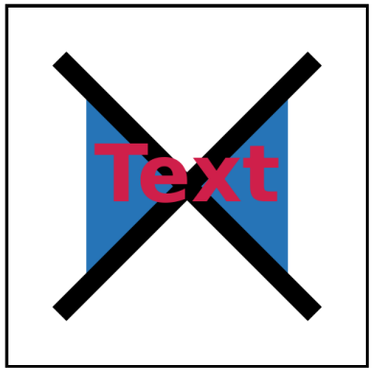
Drawings

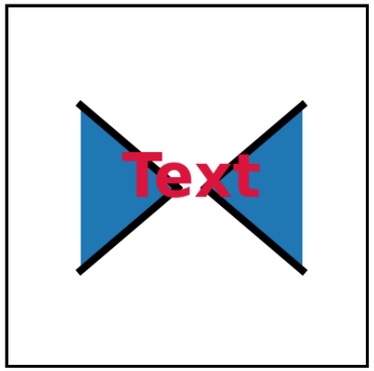
The `Path` class is useful in creating curved shapes.

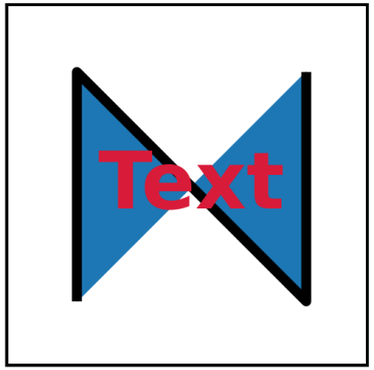
- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.

Q9: Create these two images.

→ AI solutions (AI-know, ChatGPT, Gemini)





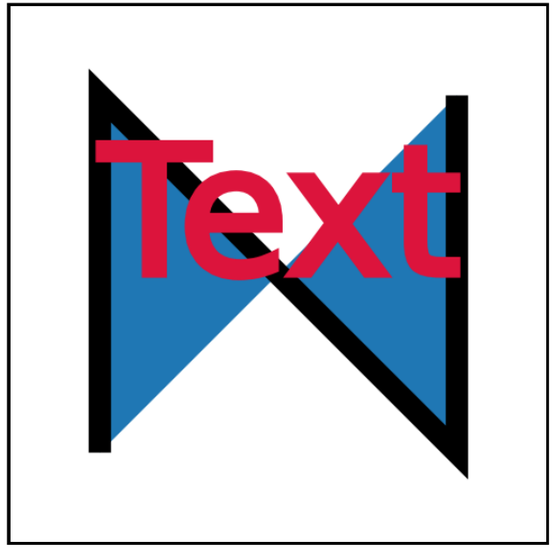


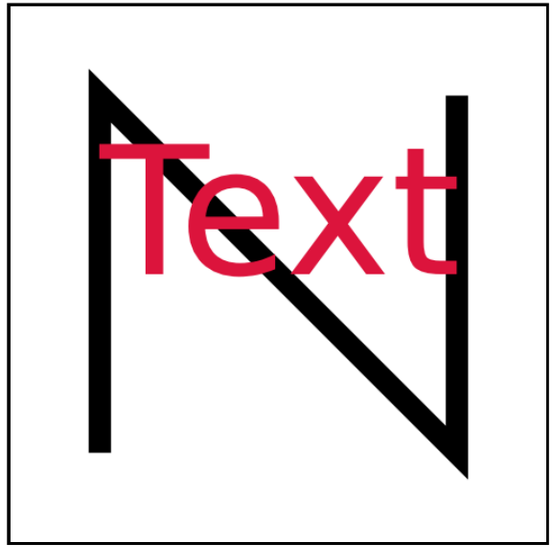
Drawings

The `Path` class is useful in creating curved shapes.

- `path = Path([(0, 0), (1, 1)])` creates a line from $(0, 0)$ to $(1, 1)$.
- `path = TextPath((0, 0), 'ABC')` creates a curve drawing 'ABC'.
- `patch = ax.add_patch(PathPatch(path, ...))` represents the curve.

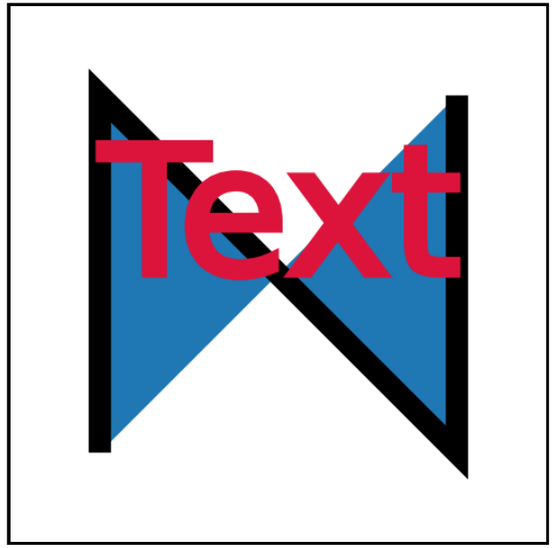
Q9: Create these two images.

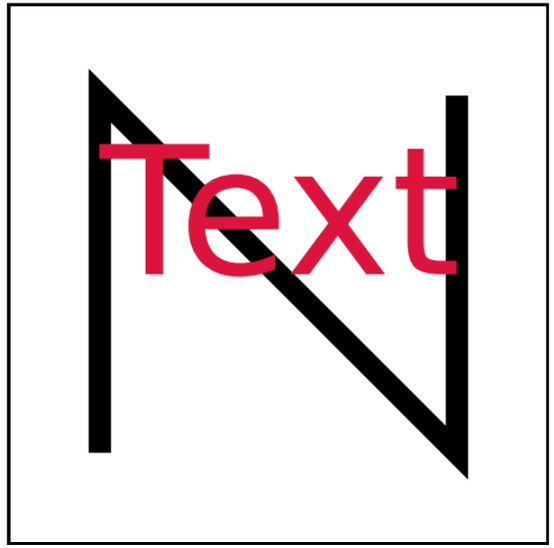




Drawings

```
172. if __name__ == '__main__':
173.     vis = Visual.square(dpi=500)
174.     vis.set_boundary(1.5)
175.     N = vis.ax.add_patch(PathPatch(
176.         path=Path([(-1, -1), (-1, 1), (1, -1), (1, 1)]),
177.         lw=3,
178.     ))
179.     T = vis.ax.add_patch(PathPatch(
180.         path=TextPath(xy=(-1, 0), s='Text', size=1),
181.         color='crimson',
182.     ))
183.     vis.new_frame()
184.     N.set_fill(False)
185.     T.set_lw(0)
186.     vis.new_frame()
```





Drawings

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

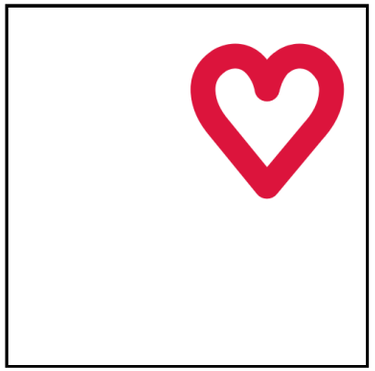
- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Q10: Create these three images.



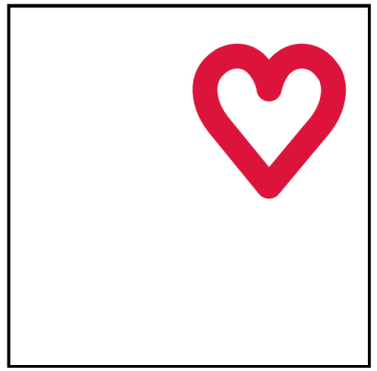
Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Q10: Create these three images.

→ AI solutions (AI-know, ChatGPT, Gemini)



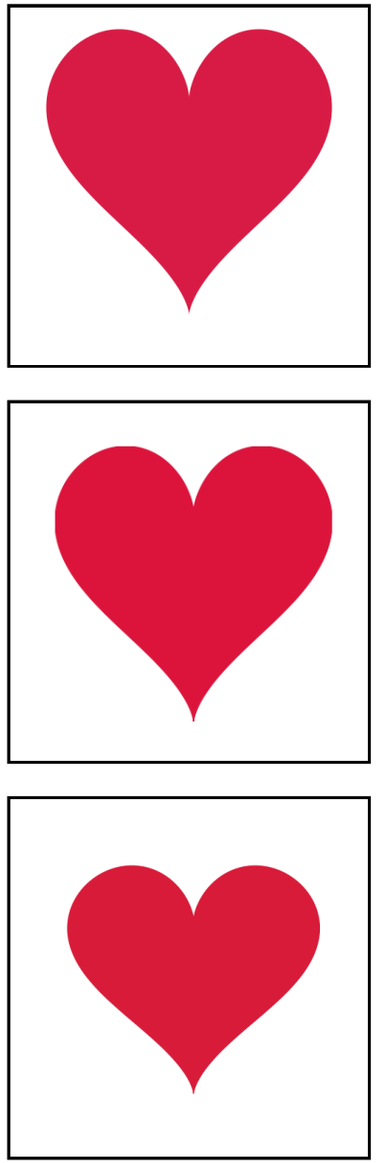
Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Q10: Create these three images.

→ AI solutions (AI-know, ChatGPT, Gemini)

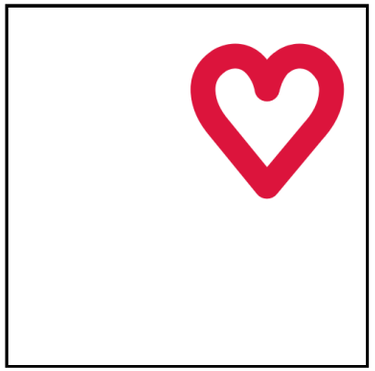


Drawings

Aligning `TextPath` is difficult because of its default anchor, but alternatives exist.

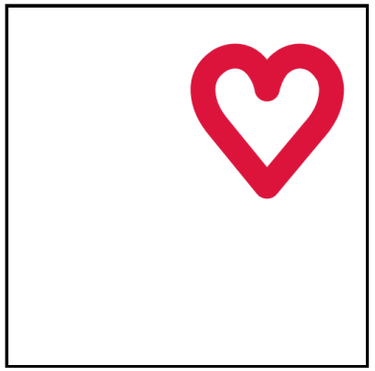
- Using `matplotlib.transforms` is the recommended way, but not covered here.
- `path.vertices` gives acces to the vertices of the curve as a `numpy` array.
- `bbox = path.get_extents()` provides a *bounding box* containing the text.
- `bbox` is an element of `matplotlib.transforms.Bbox`, with useful properties.
- `Path(a*path.vertices + b, path.codes)` linearly transforms the curve.

Q10: Create these three images.



Drawings

```
if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    path = TextPath(xy=(0, 0), s='\u2665', size=1)
    patch = vis.ax.add_patch(PathPatch(path=path))
    patch.set(lw=5, color='crimson', fill=False, joinstyle='round')
    vis.new_frame()
    bbox = path.get_extents()
    vertices = (path.vertices - (bbox.p0 + bbox.p1)/2)/bbox.size
    path = Path(vertices=vertices, codes=path.codes)
    patch.set(path=path)
    vis.new_frame()
    vis.set_boundary(0.5)
    patch.set(lw=0, fill=True)
    vis.new_frame()
```



Drawing a heart

Drawing a heart

The solution of **Q10** includes the following code which automatically adds a `patch` accessed using the `heart` attribute of the class.

Drawing a heart

The solution of **Q10** includes the following code which automatically adds a `patch` accessed using the `heart` attribute of the class.

```
    def add_heart(self, size=1, *args, **kwargs):
        path = TextPath(xy=(0, 0), s='\u2665', size=1)
        bbox = path.get_extents()
        vertices = size*(path.vertices - (bbox.p0 + bbox.p1)/2)/bbox.size
        path = Path(vertices=vertices, codes=path.codes)
        self.heart = vis.ax.add_patch(PathPatch(path=path, *args, **kwargs))
```

Drawing a heart

The solution of **Q10** includes the following code which automatically adds a `patch` accessed using the `heart` attribute of the class.

```
    def add_heart(self, size=1, *args, **kwargs):
        path = TextPath(xy=(0, 0), s='\u2665', size=1)
        bbox = path.get_extents()
        vertices = size*(path.vertices - (bbox.p0 + bbox.p1)/2)/bbox.size ........ resizes the vertices around the origin
        path = Path(vertices=vertices, codes=path.codes)
        self.heart = vis.ax.add_patch(PathPatch(path=path, *args, **kwargs))
```

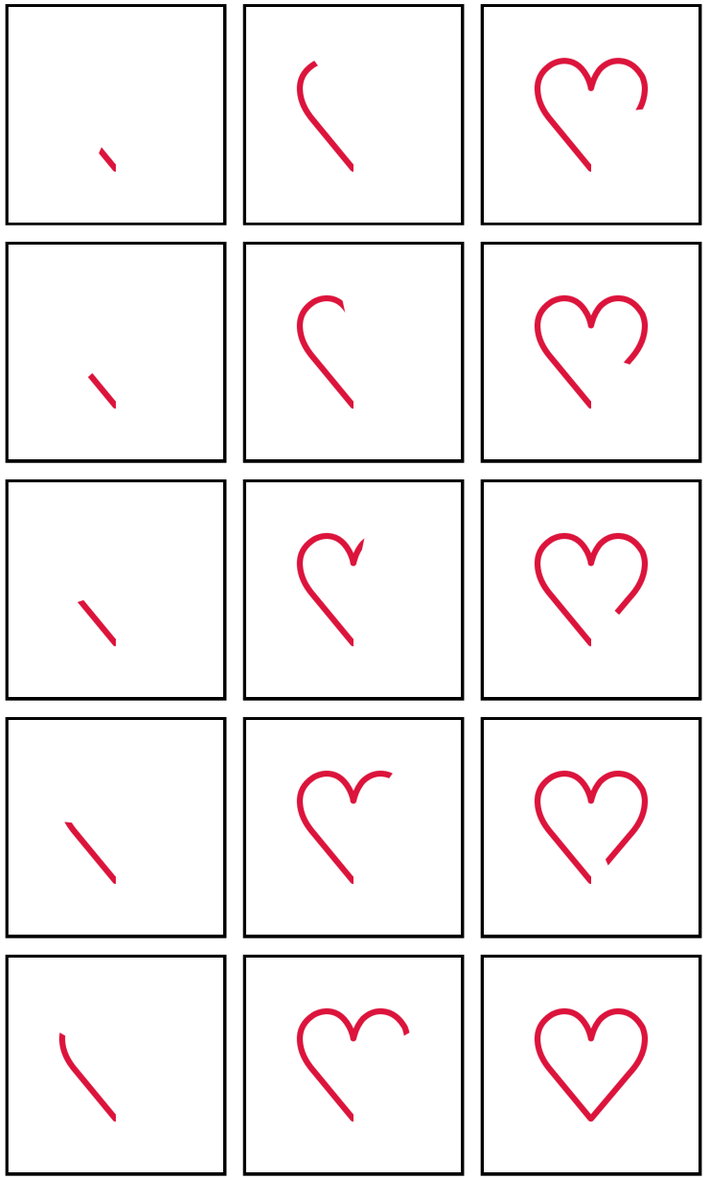
Drawing a heart

Drawing a heart

Q11: Create the drawn heart effect.

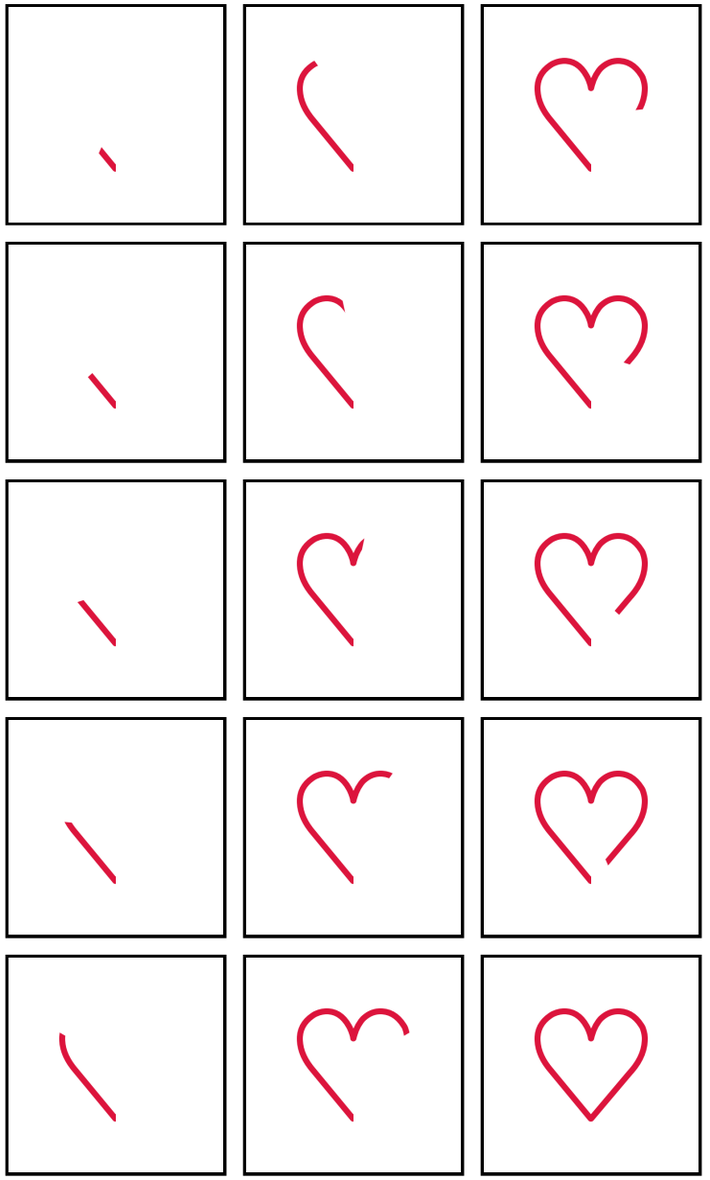
Drawing a heart

Q11: Create the drawn heart effect.



Drawing a heart

```
    def draw_heart(self, duration, *args, **kwargs):
        n_steps = self.duration_to_number(duration)
        wedge = self.ax.add_patch(
            Wedge((0, 0), 1, 270, 270, visible=False)
        )
        self.heart.set_clip_path(wedge)
        for index in range(n_steps):
            wedge.set_theta1(270 - 360*(1 + index)/n_steps)
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.add_heart(ec='crimson', lw=2, fill=False, joinstyle='round')
    vis.draw_heart(0.5)
```



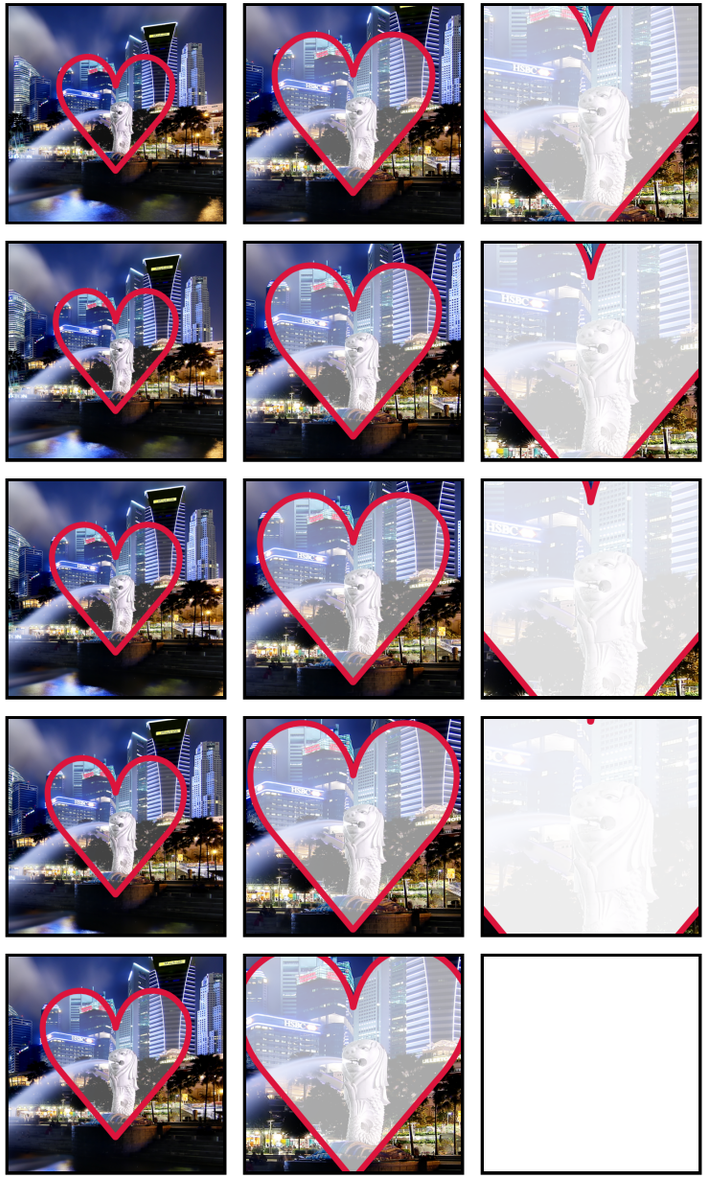
Zooming effect

Zooming effect

Q12: Create a zooming and fading effect.

Zooming effect

Q12: Create a zooming and fading effect.



Zooming effect

```
    def zoom_in(self, duration, zoom=0.5):
        n_steps = self.duration_to_number(duration)
        for index in range(n_steps):
            ratio = (1 + index)/n_steps
            self.heart.set_fc((1, 1, 1, ratio))
            self.set_boundary(1 - ratio*(1 - zoom))
            self.new_frame()



if __name__ == '__main__':
    vis = Visual.square(dpi=500)
    vis.set_boundary()
    vis.add_image(filename='singapore.jpg', shift=0.4)
    vis.add_heart(ec='crimson', lw=2, joinstyle='round')
    vis.zoom_in(0.5, zoom=0.2)
```

Table of contents

- Extending the Figure class
- Importing images
- Creating shapes
- Creating drawings
- Conclusion

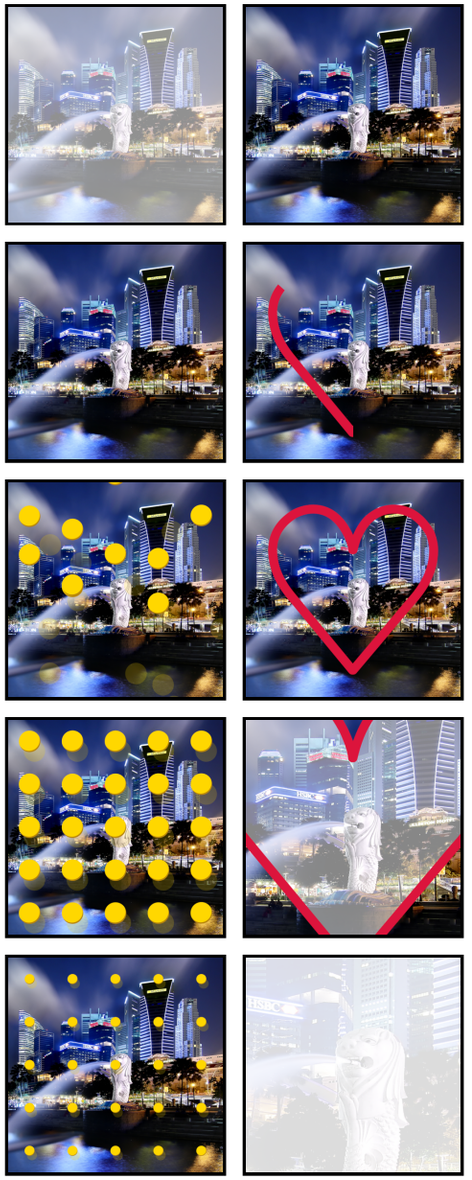
Final video

Final video

Q13: Combine all functions to create the original video.

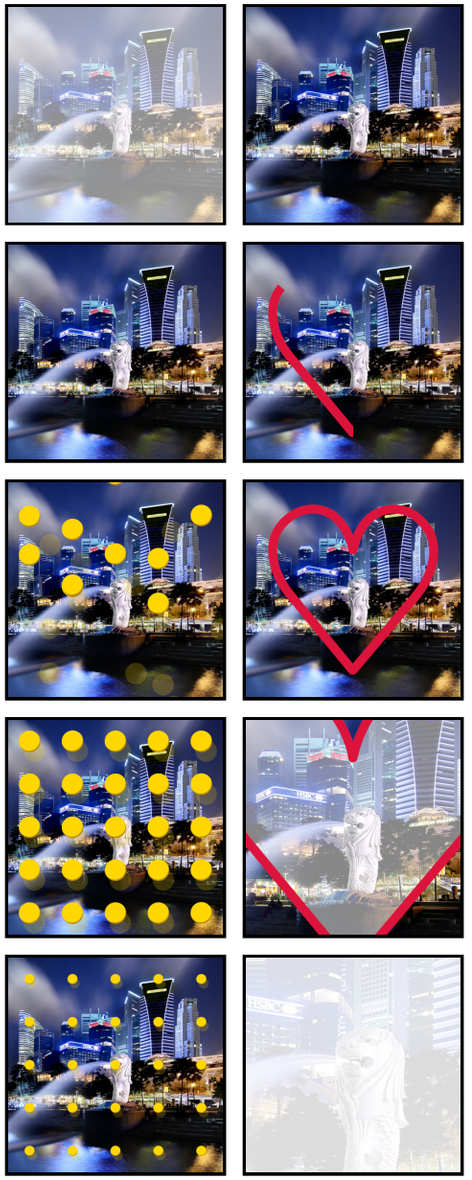
Final video

Q13: Combine all functions to create the original video.



Final video

```
if __name__ == '__main__':
    vis = Visual.square(dpi=1000)
    vis.set_boundary()
    vis.wait(duration=0.5)
    vis.add_image(filename='singapore.jpg', shift=0.4)
    vis.image_appear(duration=1)
    vis.wait(duration=0.5)
    vis.make_spheres(number=5, color='gold', dark='darkgoldenrod')
    vis.drop_spheres(duration=3)
    vis.schrink_spheres(duration=1)
    vis.wait(duration=0.5)
    vis.add_heart(size=1.5)
    vis.heart.set(ec='crimson', fc=4*[0], lw=3, joinstyle='round')
    vis.draw_heart(duration=1)
    vis.wait(duration=0.5)
    vis.zoom_in(duration=1, zoom=0.2)
    vis.make_video()
```

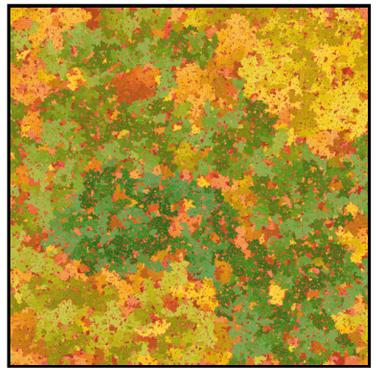


Final comments

Final comments

Hopefully, this presentation helped understand `matplotlib` and its potential.

Final comments



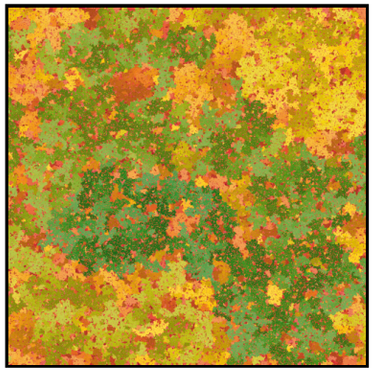
Hopefully, this presentation helped understand `matplotlib` and its potential.

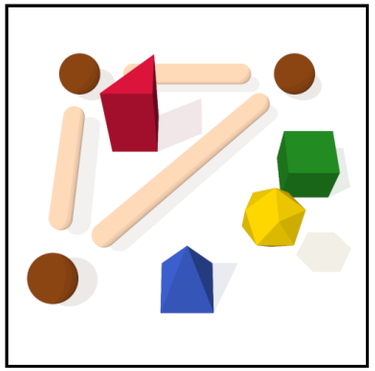
- Create detailed images pixel by pixel.

Final comments

Hopefully, this presentation helped understand `matplotlib` and its potential.

- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.

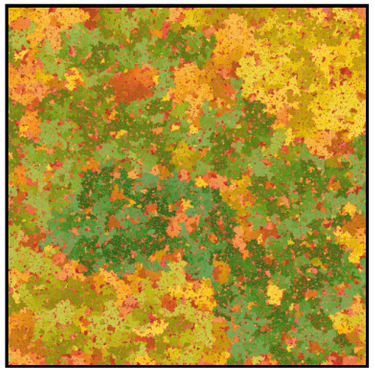


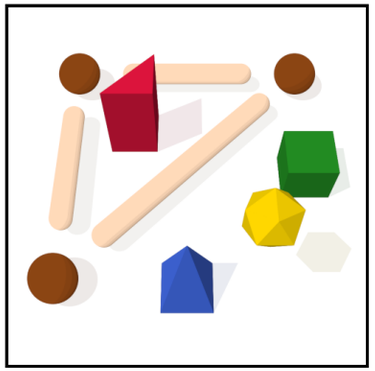


Final comments

Hopefully, this presentation helped understand `matplotlib` and its potential.

- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.
- Combine texts and curves for personalized designs.



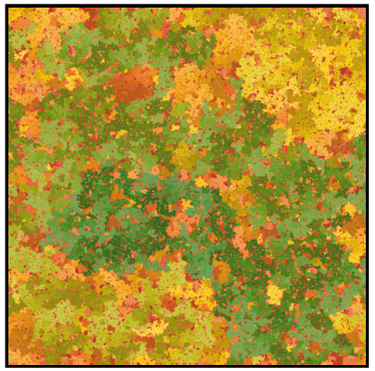


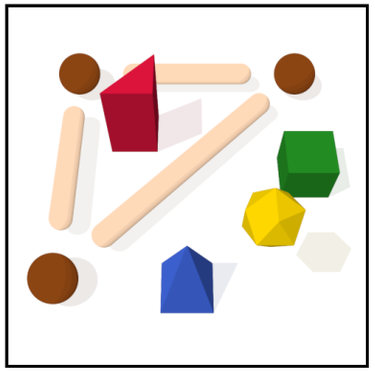
Final comments

Hopefully, this presentation helped understand `matplotlib` and its potential.

- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.
- Combine texts and curves for personalized designs.

This package also contains other useful tools not covered here.





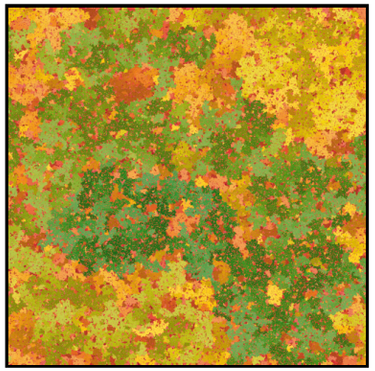
Final comments

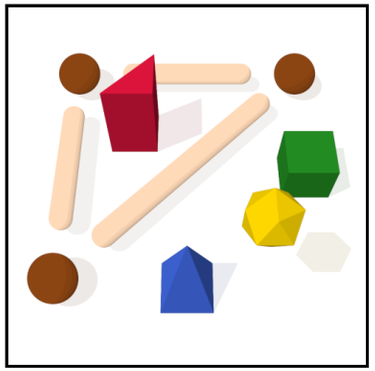
Hopefully, this presentation helped understand `matplotlib` and its potential.

- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.
- Combine texts and curves for personalized designs.

This package also contains other useful tools not covered here.

- `matplotlib.transforms` for moving shapes around.





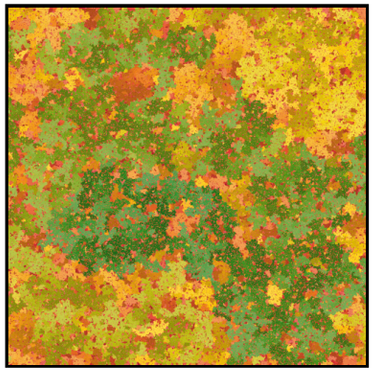
Final comments

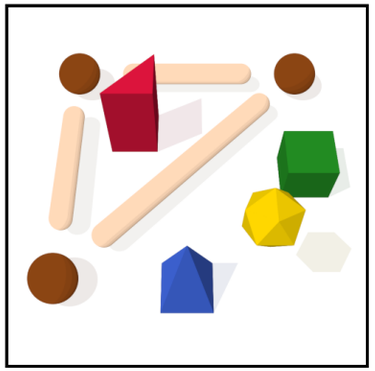
Hopefully, this presentation helped understand `matplotlib` and its potential.

- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.
- Combine texts and curves for personalized designs.

This package also contains other useful tools not covered here.

- `matplotlib.transforms` for moving shapes around.
- `matplotlib.colors` for creating smooth color transitions.





Final comments

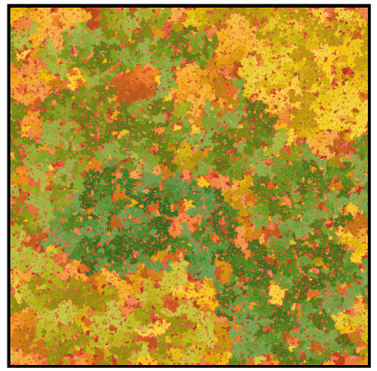
Hopefully, this presentation helped understand `matplotlib` and its potential.

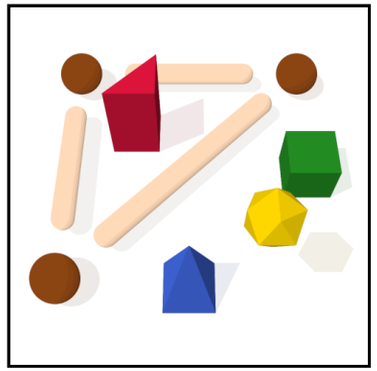
- Create detailed images pixel by pixel.
- Use shapes to represent 3D effects.
- Combine texts and curves for personalized designs.

This package also contains other useful tools not covered here.

- `matplotlib.transforms` for moving shapes around.
- `matplotlib.colors` for creating smooth color transitions.

And so much more still remain to be discovered and applied!





Thank you!
Thank you!
Thank you!
Maybank
HSBC
HSBC
ONE
FULLERTON
Thank you!
Thank you!
Thank you!
Thank you!
Thank you!
Thank you!
Thank you!
Thank you!
Thank you!