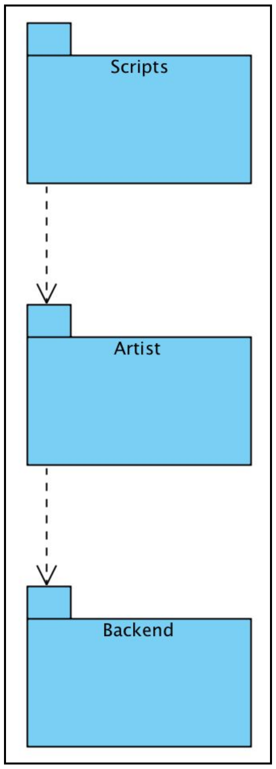
Matplotlib

Deliverable 2: Reverse Engineering Report

By: Team Legend (Team 4)

Table of Contents

System Architecture Diagrams



High-level Architecture Diagram of Matplotlib

Commentary on Architecture

The architecture of matplotlib is designed as a stack of three subsystems. Backend is at the very bottom of the architectural stack, Artist in the middle and Scripting at the top. Each subsystem only knows and interacts with the subsystems underneath it in the stack. Due to this, the degree of coupling between modules is relatively low, as a lower module will never be reliant on a higher module. However, the degree of coupling within the modules themselves is relatively high, as many classes within the same module are interdependent on each other. We believe this is a solid conceptual design, as the main components of matplotlib remain fairly modular, while still taking advantage of coupling and inheritance in the lower level implementations.

The lowest level of the designed stack is the Backend subsystem, which due to it being at the bottom of the stack, is entirely independent and isolated. There are abstract backend classes in:

/lib/matplotlib/backend_bases.py, /lib/matplotlib/backend_managers.py

Which prescribe the required methods for the concrete RendererBase and

FigureCanvasBase classes in each concrete backend implementation. Each concrete backend is designed to be modular and can be switched out based on user preference. There are two general types of implemented backends: user interface and hard copy backends. The user interface implementations of backend provide a keyboard and mouse user interface to allow the user to interact with various shapes, graphs, etc. Examples of already implemented user interface backends include: wxWidgets, TK, and GTK. The hard copy backends allow matplotlib to work with and generate plots onto hard copy types with no user interface. Some examples include: PDF (/lib/matplotlib/backends/backend_pdf.py), PNG, PS. Both types of backends are used to render a figure and generate canvases for an Artist to work with.

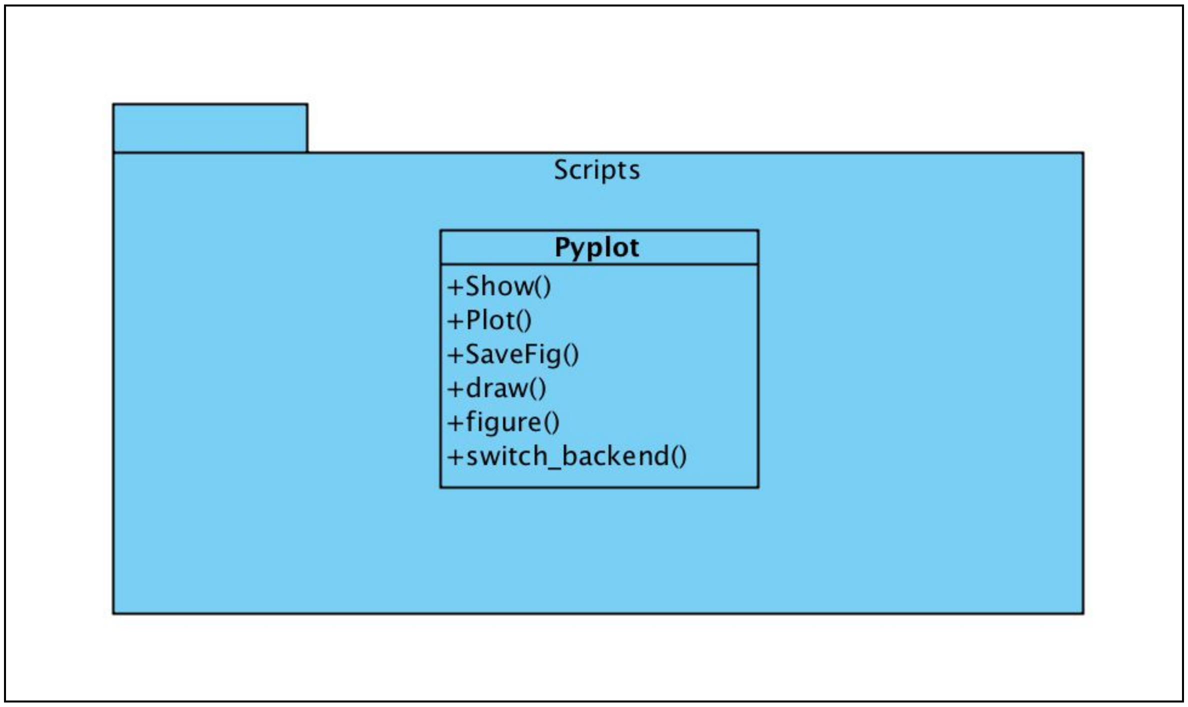
The next layer upwards, the Artist layer, contains classes that deal with actually creating the image. Each image (plot) is made up of various artists (a figure, shape, curve, graph, axis, etc.) that inherit from the abstract Artist class. Each artist class then uses the renderer API provided by the selected backend implementation to draw on the canvas in the current implementation. The artist classes only interact with the canvas through the renderer API, the actual implementation of the backend does not change the artist classes in any way. To make an image initially the Artist layer will create a Figure, that will then build the image by adding additional artist classes to draw each part of the image. Once a canvas and renderer are supplied, each subclass of Artist can begin to draw itself on the canvas, creating a full image for the user.

The scripting layer uses the two layers beneath it to give users the pyplot interface for matplotlib. It handles an API to make working with matplotlib easier and more efficient for the user. The scripting layer is entirely optional and in no way adds to matplotlib's main functionality. This API is very useful for developers, scientists, and analysts, as it provides an intuitive way to interact with matplotlib (similar to MATLAB).

Although the conceptual architecture and design of matplotlib is fairly intuitive and decently organized, we feel that this architecture was poorly organized in the actual project. Files in each of the three subsystems seem to be sporadically placed through different sub-packages (with the majority of classes for each conceptual layer in lib/matplotlib) with poor organization. Currently, if you were given a class and which subsystem it belongs to, that would still not be enough information to actually pinpoint the location of the file containing it. We believe that it would be significantly better if they had created packages pertaining to each subsystem described above in the system architecture, as this would more directly relate to their design of matplotlib. Of course the file structure doesn't have a large impact on the matplotlib application itself, but it would be a significant improvement for developers if it was organized in a more structural matter.

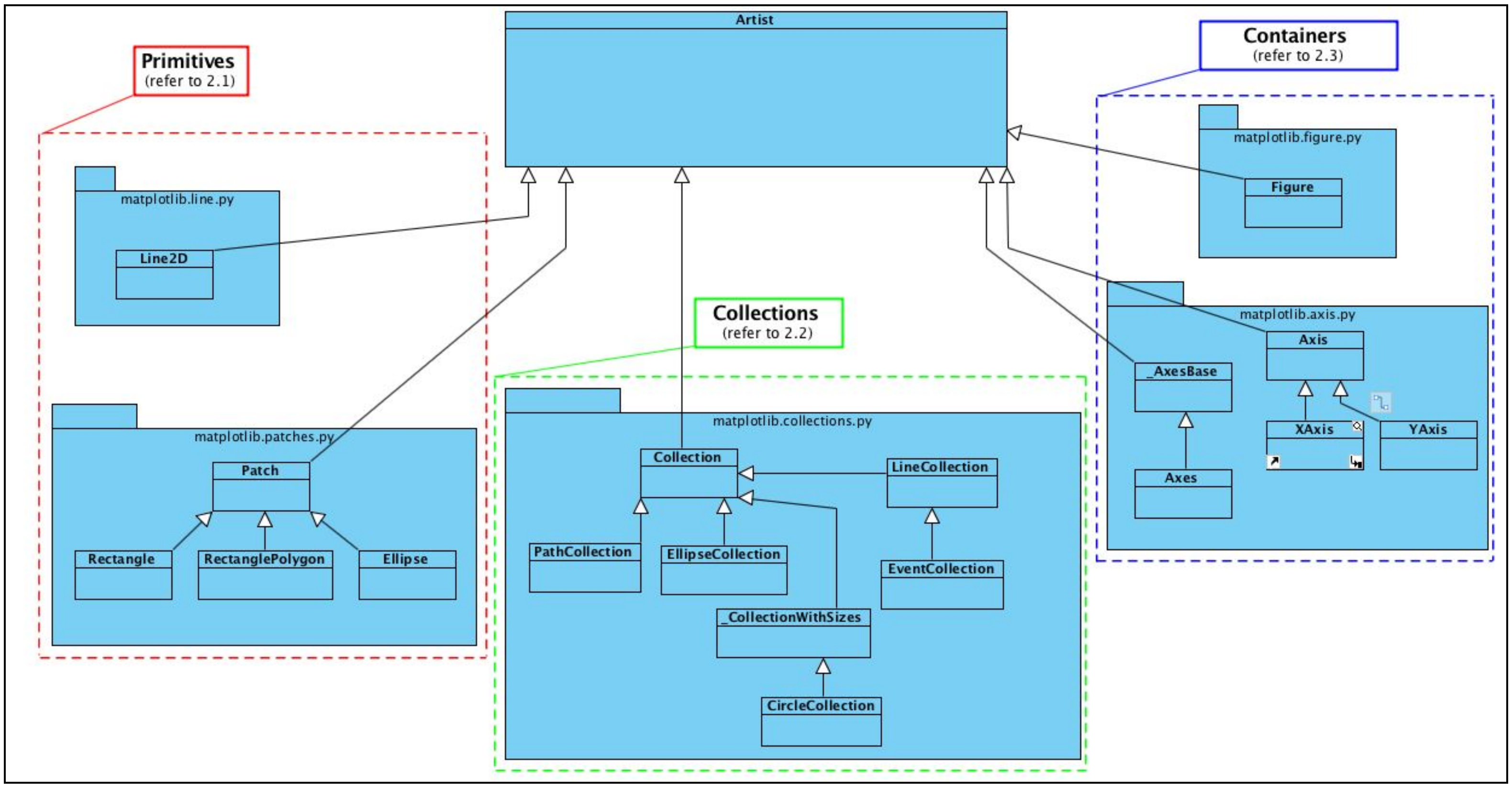
Scripts

This subsystem provides scripts like pyplot, which make easier for non-programmers to use matplotlib without having to make complicated calls artist and backend system. Matplotlib can fully function without Script subsystem.



Artist

In the UML class diagram below, there are three components that make up the artist layer - Primitives, Collections, and Containers. Primitives and Containers are the two main components in this layer



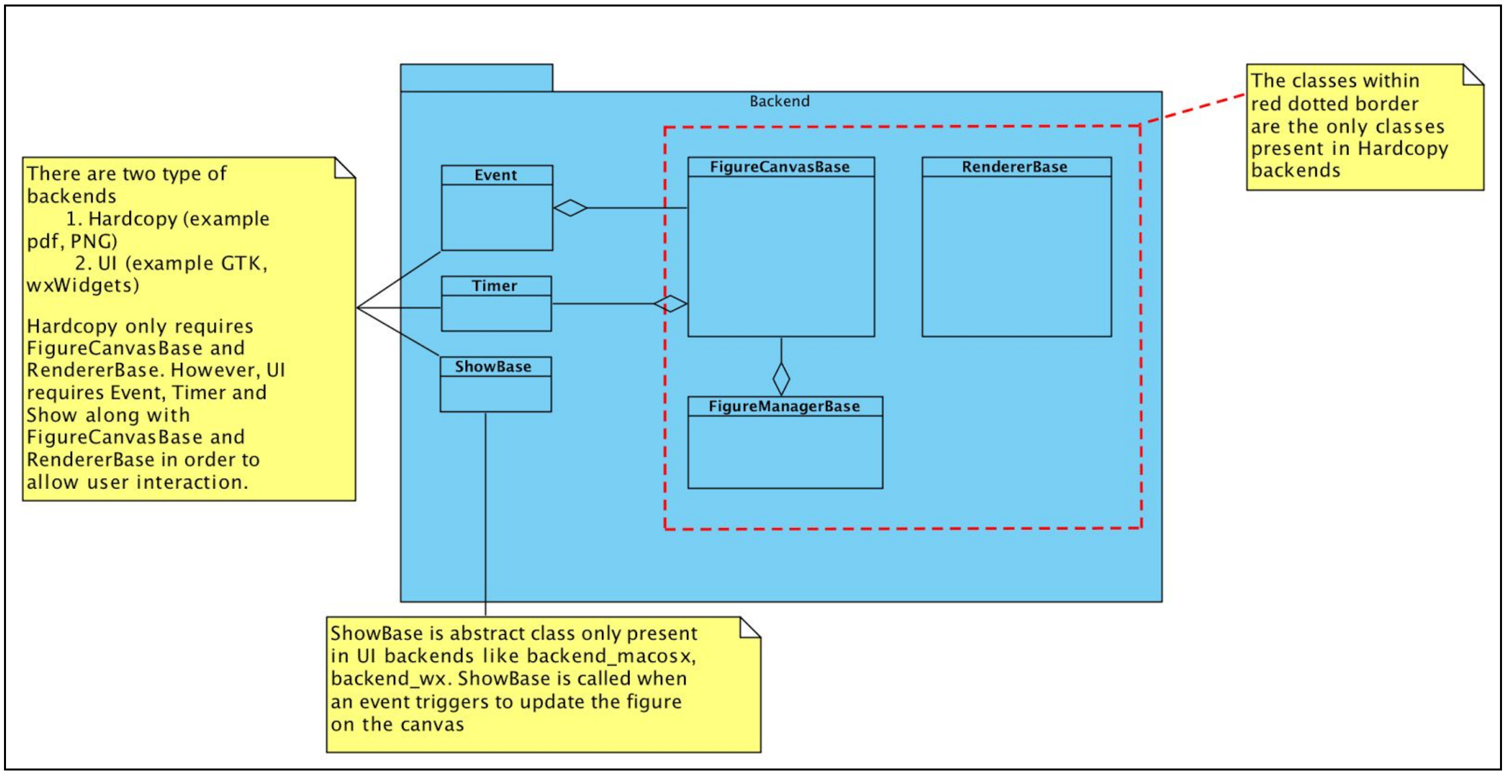
2.1 The primitives include classes such as Line2D and Patch (shapes) that have subclasses such as Rectangle, RectanglePolygon, Ellipse, etc. They are subclasses of the artist and represent the graphical objects that are painted on the canvas.

2.2 This is another component of the artist layer. The collections component makes drawing similar objects much easier. The classes for this component includes PathCollection, EllipseCollection, CircleCollection, EventCollection, etc.

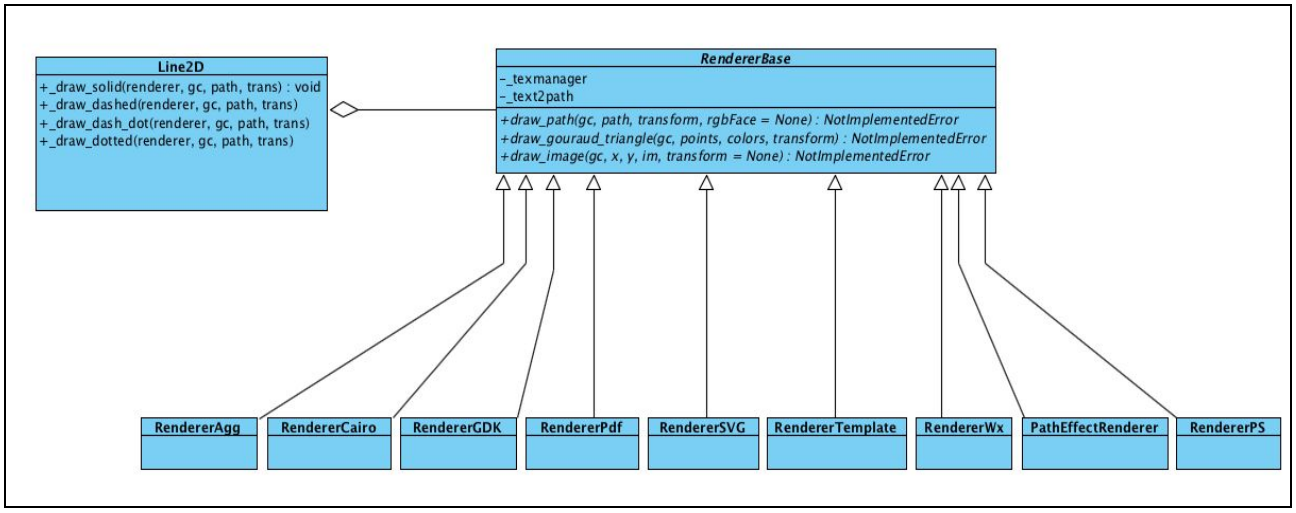
2.3 The containers represent the location of the Primitives. It includes classes such as Figure, Axis(with subclasses XAxis and YAxis), and Axes. Usually, an instance of Figure is created which is then used to create instances of other classes such as the Axes. In the case that an instance of the Axes is created, the user is then able to create other primitives using the Axes object.

Backend

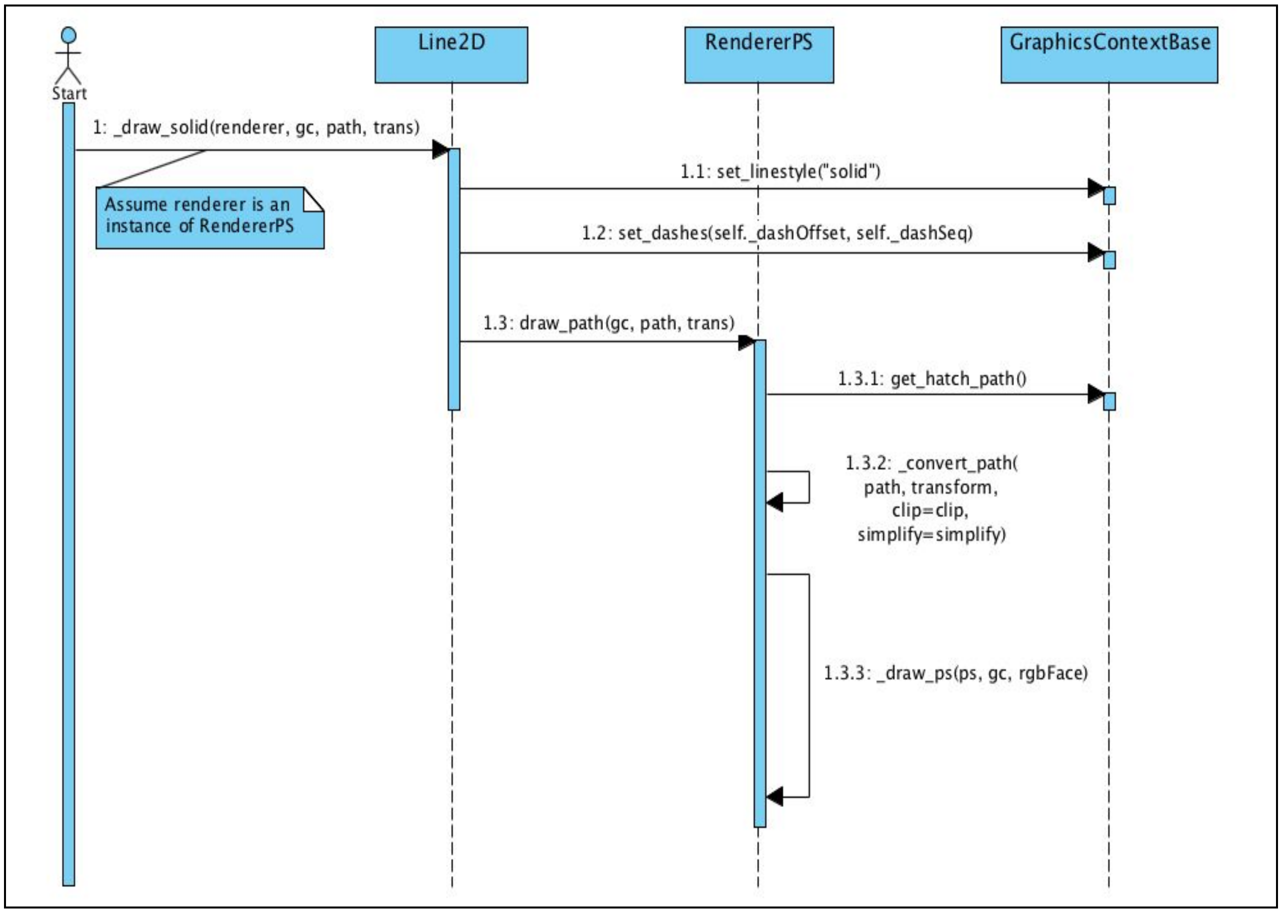
The following UML shows the general structure of the backends


Backend
Event
Timer
ShowBase
FigureCanvasBase
RendererBase
FigureManagerBase
There are two type of backends
1. Hardcopy (example pdf, PNG)
2. UI (example GTK, wxWidgets)
Hardcopy only requires FigureCanvasBase and RendererBase. However, UI requires Event, Timer and Show along with FigureCanvasBase and RendererBase in order to allow user interaction.
The classes within red dotted border are the only classes present in Hardcopy backends
ShowBase is abstract class only present in UI backends like backend_macosx, backend_wx. ShowBase is called when an event triggers to update the figure on the canvas

Design Pattern 1 – Strategy Design Pattern


Line2D
+_draw_solid(renderer, gc, path, trans) : void
+_draw_dashed(renderer, gc, path, trans)
+_draw_dash_dot(renderer, gc, path, trans)
+_draw_dotted(renderer, gc, path, trans)
RendererBase
-_texmanager
-_text2path
+draw_path(gc, path, transform, rgbFace = None) : NotImplementedError
+draw_gouraud_triangle(gc, points, colors, transform) : NotImplementedError
+draw_image(gc, x, y, im, transform = None) : NotImplementedError
RendererAgg
RendererCairo
RendererGDK
RendererPdf
RendererSVG
RendererTemplate
RendererWx
PathEffectRenderer
RendererPS

UML Diagram for Strategy Design Pattern


Start
Line2D
RendererPS
GraphicsContextBase
1: _draw_solid(renderer, gc, path, trans)
Assume renderer is an instance of RendererPS
1.1: set_linestyle("solid")
1.2: set_dashes(self._dashOffset, self._dashSeq)
1.3: draw_path(gc, path, trans)
1.3.1: get_hatch_path()
1.3.2: _convert_path(path, transform, clip=clip, simplify=simplify)
1.3.3: _draw_ps(ps, gc, rgbFace)

Sequence Diagram of Strategy Design Pattern

Description

In the file /matplotlib/lib/matplotlib/backend_bases.py, there is an abstract class called RendererBase, the parent of 9 classes:

1. RendererAgg
2. RendererCairo
3. RendererGDK
4. RendererPdf
5. RendererSVG
6. RendererTemplate
7. RendererWx
8. PathEffectRenderer
9. RendererPS

There are 3 abstract methods in RendererBase:

1. draw_path(gc, path, transform, rgbFace = None)
2. draw_gouraud_triangle(gc, points, colors, transform)
3. draw_image(gc, x, y, im, transform = None)

Each of the abstract methods in RendererBase returns NotImplementedError to force the implementation of these methods in its child classes. The abstract class used in Python is similar to the interface in Java as Python allows inheritance from more than one superclass.

In the file /matplotlib/lib/matplotlib/lines.py, there is a class called Line2D, which has an aggregation relationship with Axis. In this case, there are four methods in Line2D that calls draw_path:

1. _draw_solid(renderer, gc, path, trans)
2. _draw_dashed(renderer, gc, path, trans)
3. _draw_dash_dot(renderer, gc, path, trans)
4. _draw_dotted(renderer, gc, path, trans)

All of these four methods in the class Line2D take renderer as one of the inputs. Thus, the line renderer.draw_path(...) may execute different blocks of code depending on the subclass of RendererBase that renderer refers to. For example, in the sequence diagram, when renderer is an instance of RendererPS, the draw_path(...) method in RendererPS will be executed. However, if renderer refers to another subclass of RendererBase, then a different draw_path(...) method will be executed. This matches the Strategy Design Pattern.

Link to Corresponding code

class RendererBase : /matplotlib/lib/matplotlib/backend_bases.py line 194

- draw_path(...) : line 234

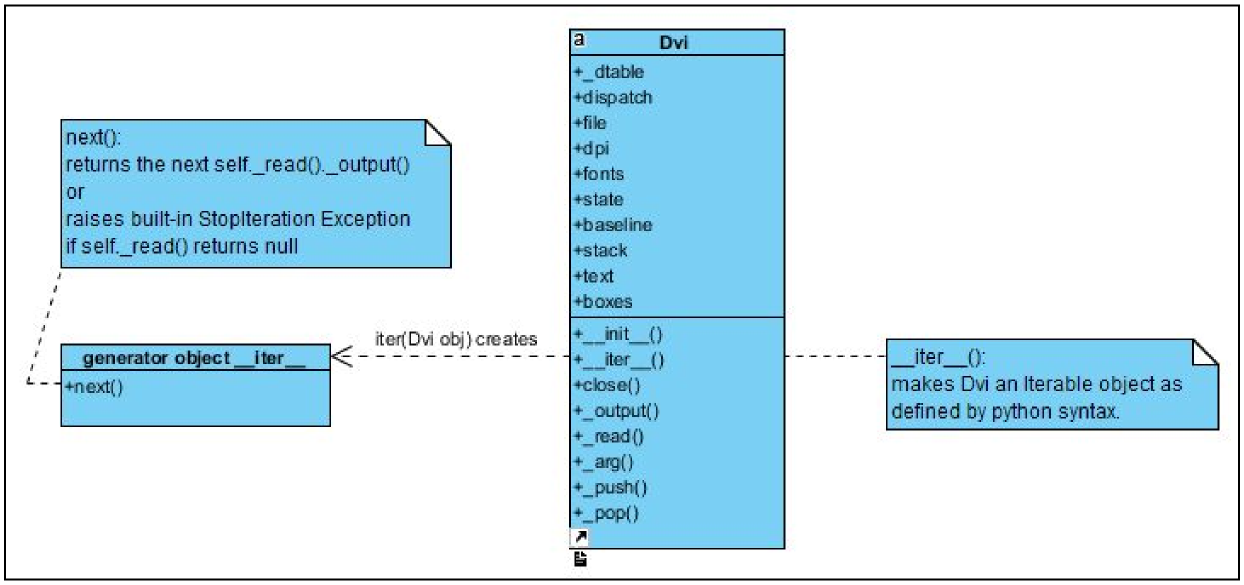
class RendererPS: /matplotlib/lib/matplotlib/backends/backend_ps.py line 173

- draw_path(...) : line 501

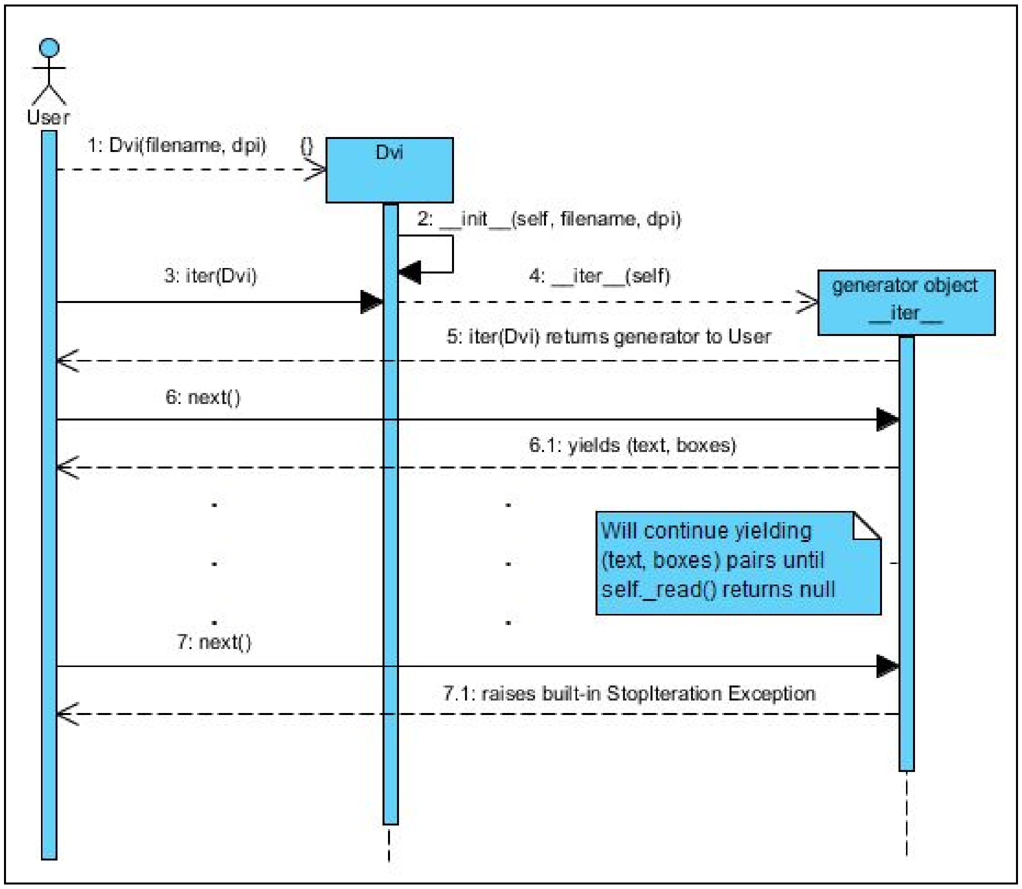
class Line2D: /matplotlib/lib/matplotlib/lines.py line 232

- _draw_solid(...) : line 1245

Design Pattern 2 – Iterator Design Pattern


Dvi
+_dtable
+dispatch
+file
+dpi
+fonts
+state
+baseline
+stack
+text
+boxes
+__init__()
+__iter__()
+close()
+_output()
+_read()
+_arg()
+_push()
+_pop()
next():
returns the next self._read()._output()
or
raises built-in StopIteration Exception
if self._read() returns null
generator object __iter__
+next()
iter(Dvi obj) creates
__iter__():
makes Dvi an Iterable object as
defined by python syntax.

Dvi Class UML (Note: non-informative/relevant methods removed)


User
Dvi
1: Dvi(filename, dpi)
2: __init__(self, filename, dpi)
3: iter(Dvi)
4: __iter__(self)
generator object __iter__
5: iter(Dvi) returns generator to User
6: next()
6.1: yields (text, boxes)
Will continue yielding (text, boxes) pairs until self._read() returns null
7: next()
7.1: raises built-in StopIteration Exception

Sequence Diagram of Iterator Design Pattern for class Dvi

Description

In the file /matplotlib/lib/matplotlib/dviread.py, there is a class called Dvi, that as described in its docstring is: “A reader for a dvi ("device-independent") file, as produced by TeX. The current implementation can only iterate through pages in order, ... This class can be used as a context manager to close the underlying file upon exit. Pages can be read via iteration.” (See /matplotlib/lib/matplotlib/dviread.py lines 177-182).

The Iterator Design Pattern is present in the code via the __iter__(self) method starting on line 224, which is described by the docstring as a method that will:

> “Iterate through the pages of the file. Returns (text, boxes) pairs, where:
>
> text is a list of (x, y, fontnum, glyphnum, width) tuples
>
> boxes is a list of (x, y, height, width) tuples”

(See /matplotlib/lib/matplotlib/dviread.py lines 226-230).

This particular implementation of the Iterator Design Pattern uses the python specific generator variant, making use of the yield statement. Specifically when called the method returns a generator (type of iterator) object that will yield the next (text, boxes) tuple when called. In this case the yield statement takes the place of the more standard separately defined next() method; in python yield will keep returning the next element as defined in the generator when next() is called.

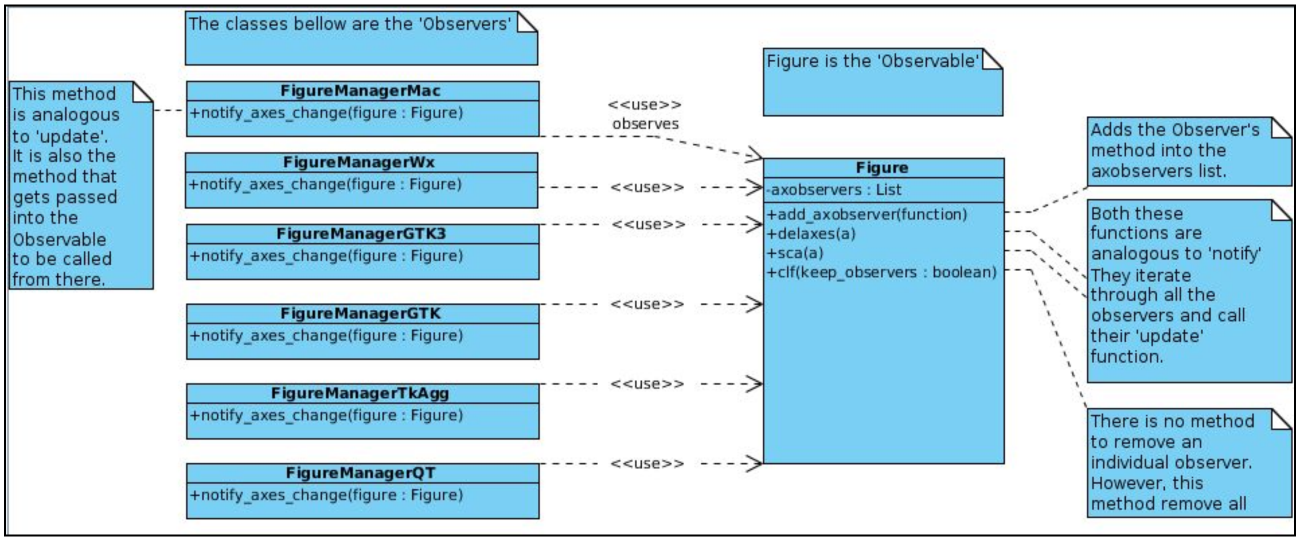
When iter() (either directly or in a for loop) is called it will return an iterator object (generator object __iter__). Then when next() is first called (either directly or in a for loop) it will execute the __iter__ method from the beginning of the method entering the while loop, setting have_page = self._read(), then if a page was read it will yield (return) self._output(), otherwise it will break and automatically raise a built-in StopIteration Exception. On each subsequent call to next() it will continue from the next point in the code after the yield statement, until it breaks and then will only raise a StopIteration Exception on each subsequent call.

By containing the __iter__(self) method it is the equivalent of inheriting the iterable interface, making the Dvi class an Iterable class as defined by the design pattern. When iter() is called on a Dvi, it returns the generator object __iter__ which is equivalent to an object implementing the Iterator interface, making the generator object an Iterator class as defined by the design pattern.

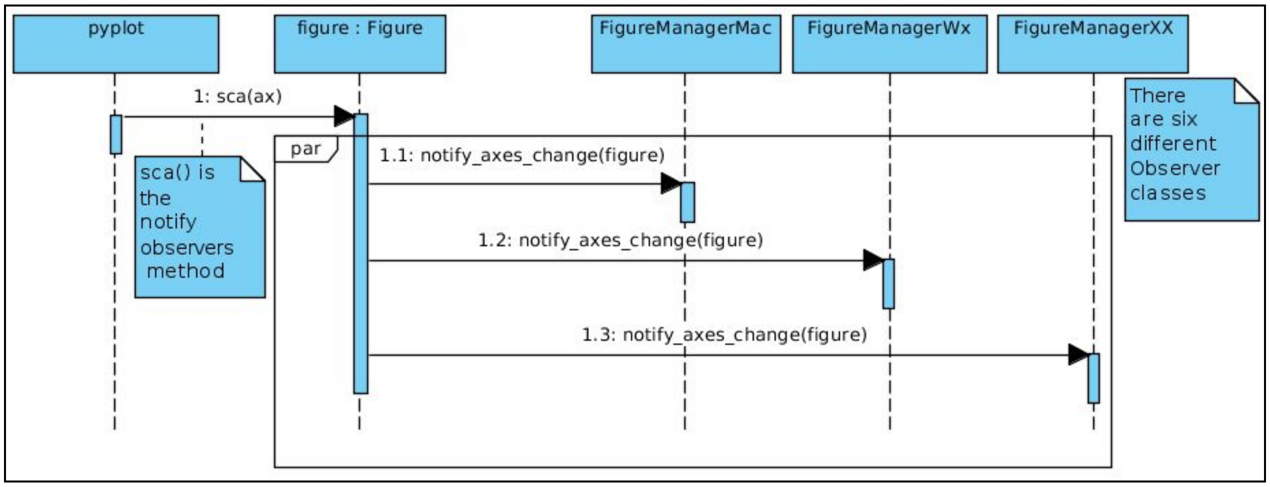
Link to Corresponding code

class Dvi : /matplotlib/lib/matplotlib/dviread.py line 175 → __iter__(self) : line 224

Design Pattern 3 – Observable Design Pattern


The classes bellow are the 'Observers'
This method is analogous to 'update'. It is also the method that gets passed into the Observable to be called from there.
FigureManagerMac
+notify_axes_change(figure : Figure)
FigureManagerWx
+notify_axes_change(figure : Figure)
FigureManagerGTK3
+notify_axes_change(figure : Figure)
FigureManagerGTK
+notify_axes_change(figure : Figure)
FigureManagerTkAgg
+notify_axes_change(figure : Figure)
FigureManagerQT
+notify_axes_change(figure : Figure)
<<use>> observes
<<use>>
Figure is the 'Observable'
Figure
-axobservers : List
+add_axobserver(function)
+delaxes(a)
+sca(a)
+clf(keep_observers : boolean)
Adds the Observer's method into the axobservers list.
Both these functions are analogous to 'notify' They iterate through all the observers and call their 'update' function.
There is no method to remove an individual observer. However, this method remove all

Structural UML diagram for the Observer Design Pattern found in Matplotlib


pyplot
figure : Figure
FigureManagerMac
FigureManagerWx
FigureManagerXX
1: sca(ax)
par
sca() is the notify observers method
1.1: notify_axes_change(figure)
1.2: notify_axes_change(figure)
1.3: notify_axes_change(figure)
There are six different Observer classes

Sequence UML diagram for notifying the Observers found in Matplotlib

Description

The way the design pattern was implemented in Matplotlib is quite different from what is expected. It retains the base concept: if an object has changed, then the objects that want to observe it react to the change.

The class Figure retains a list of observers. That is not quite true, it actually retains a list of functions. Each of these functions are provided by the observer. Then when the method delaxes() or sca() is called in an instance of Figure (which implies the axes have changed) all the functions are called. These functions are analogous to update() function we usually expect in an Observer design pattern. Each function does something specific to the observer class it came from.

Link to Corresponding code

class Figure : /matplotlib/lib/matplotlib/figure.py

- add_axobserver(...) : line 1659
- delaxes(...): line 813
- sca(...): line 1563
- clf(...): line 1176

class FigureManagerMac : /matplotlib/lib/matplotlib/backends/backend_macosx.py

- notify_axes_change(figure : Figure) : line 196

class FigureManagerWx :/matplotlib/lib/matplotlib/backends/backend_ws.py

- notify_axes_change(figure : Figure) : line 1368

class FigureManagerGTK3 :/matplotlib/lib/matplotlib/backends/backend_gtk3.py

- notify_axes_change(figure : Figure) : line 459

class FigureManagerGTK :/matplotlib/lib/matplotlib/backends/backend_gtk.py

- notify_axes_change(figure : Figure) : line 604

class FigureManagerTkAgg :/matplotlib/lib/matplotlib/backends/backend_tkagg.py

- notify_axes_change(figure : Figure) : line 561

class FigureManagerQT :/matplotlib/lib/matplotlib/backends/backend_qt5x.py

- notify_axes_change(figure : Figure) : line 520

References:

The following was read to help get a better understanding of the matplotlib architecture:

Hunter, J & Droettboom, M. The Architecture of Open Source Applications Matplotlib chapter. Retrived on Feburary 10, 2017 from: http://www.aosabook.org/en/matplotlib.html (Found through recommended external documentation by matplotlib http://matplotlib.org/resources/index.html)

McGreggor, D. M. Mastering matplotlib, Chapter 2. The matplotlib Architecture (Specifically: The current matplotlib architecture, The backend layer, The artist layer, & The scripting layer).
https://www.packtpub.com/mapt/book/Big+Data+and+Business+Intelligence/9781783987542/2