1. Since Gaphor is built on GTK+ and Cairo, (Capitalize Cairo as you have done in Gaphas Documentation).

Do you have a Glossary to what GTK+ and Cairo are?

Or maybe put hyperlinks on these.

1. However, there wasn't a project that abstracted these technologies to easily create a diagramming tool.

Explain here—What you are trying to achieve using Gaphas or this is the reason why Gaphas was created…

1. Here is how it works:

Before putting the above line, maybe could explain what Canvas is?

1. <https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/>

This link has a detailed explanation to create virtual environments in different OS. I am a newbie to Python and I use Mac.

(Installing Gaphas on MacOS was not straightforward and some libraries had to be installed and took a while to get it all running. (pkg-config, cairo, libffi, gobject-introspection). The thing about documentation is to help the user with all the needed information for smoother installation).

How about steps to uninstall Gaphas?

<https://doc.rust-lang.org/book/ch01-01-installation.html>

Maybe you can refer to the above link how they have described the installation and uninstall process.

Installation for Linux and Mac

Installation for Windows if any

Uninstall

Troubleshooting

1. Gaphas was built to provide the foundational diagramming portions of [Gaphor](https://github.com/gaphor/gaphor). Since Gaphor is built on GTK+ and Cairo, [PyGObject](https://pygobject.readthedocs.io/) provides access to the GUI toolkit and [PyCairo](https://pycairo.readthedocs.io/) to the 2D graphics library. However, there wasn't a project that abstracted these technologies to easily create a diagramming tool.

Here is how it works:

* Items (Canvas items) can be added to a Canvas.
* The Canvas maintains the tree structure (parent-child relationships between items).
* A constraint solver is used to maintain item constraints and inter-item constraints.
* The item (and user) should not be bothered with things like bounding-box calculations.
* Very modular—eg., handle support could be swapped in and swapped out.
* Rendering using Cairo.

The main portions of the library include:

* Canvas - The main Canvas class (container for Items).
* items - Objects placed on a Canvas. (**please check the usage of items. It is capitalized in so many other places**).
* solver - A constraint solver to define the layout and connection of items.
* view - Responsible for the calculation of bounding boxes which is stored in a quadtree data structure for fast access.
* gtkview - A view to be used in GTK+ applications that interacts with users with tools.
* painters - The workers used to paint items.
* tools - Tools are used to handle user events (such as mouse movement and button presses).
* aspects - Provides an intermediate step between tools and items.
* **Usage**

What is the purpose of this code? A small description of what the code is and how to run it could help.

### Overview

The Canvas class (from canvas.py) acts as a container for Items (from item.py). The Items parent/child relationships are maintained here (not in the Item!).

An Item can have a set of Handles (from connector.py) which can be used to manipulate the Item (although this is not necessary). Each Item has its own coordinate system with x and y position, for example, a (0, 0) point. Item.matrix is the transformation relative to the parent item of the Item, as defined in the Canvas.

Handles can connect to Ports. A Port is a location (line or point) where a handle is allowed to connect on another Item. The process of connecting depends on the case at hand, but most often involves the creation of some sort of constraint between the Handle and the item it is connecting to (see doc/ports.txt).

**Ports.rst exists and not ports.txt**

The Canvas also contains a constraint Solver (from solver.py) that can be used to solve mathematical dependencies between Items (such as Handles that should be aligned). The constraint solver can also be used to keep constraints contained within the Item true, for example, to make sure a box maintains its rectangular shape.

View (from view.py) is used to visualize a canvas. On a View, a Tool (from tool.py) can be applied, which will handle user input like button and key presses. Painters (from painter.py) are used to do the actual drawing. This module also makes it easy to draw to other media other than a screen, such as a printer or a PDF document.

### Updating Item State

If Items need updating, it sends out an update request on the Canvas (Canvas.request\_update()). The Canvas performs an update by performing the following steps:

1. Pre-update using Item.pre\_update (context) for each item marked for update.
2. Update the Canvas-to-Item matrices, for fast transformation of coordinates from the Canvas ‘to the items' coordinate system. The c2i matrix is stored on the Item as Item.\_matrix\_c2i.
3. Solve the constraints.
4. Normalize the items by setting the coordinates of the first handle to (0, 0).
5. Update the Canvas-to-Item matrices for items that have been changed by normalization.
6. Post-update using Item.post\_update (context) for each item marked for an update, including items that have been marked during the constraint-solving step.

Gaphas attempts to do as much updating as possible in the {pre|post}\_update() methods since they are called when the application is not handling user input.

The context contains a CairoContext. This can be used, for example, to calculate the dimensions of text. One thing to keep in mind is that updating is done from the Canvas. Items should not update sub-items. After the update steps are complete, the Item should be ready to be drawn.

### Constraint solving

Constraint solving is one of the big features of this library. The Solver is able to mathematically solve these constraint rules that are applied to an item or between items. Constraints are applied to items through Variables owned by the item. An example of applying a constraint to an item is that Element items use constraints to maintain their rectangular shape. An example of applying constraints between items is to apply a constraint between a line and a box in order to connect them.

Constraints that apply to one item are straight forward, as all variables live in the same coordinate system of the item. The variables, like the Handle's x and y coordinate, can simply be put in a constraint.

When two items are connected to each other and constraints are created, a problem shows up: variables live in separate coordinate systems. In order to overcome this problem, a Projection (from solver.py) has been defined. With a Projection instance, a variable can be "projected" on another coordinate system. In this case, the Canvas' coordinate system is used when two items are connected to each other.

### Drawing

Drawing is done by the View. All items marked for redraw, the items that have been updated, will be drawn in the order in which they reside in the Canvas. The order starts with the first root item, then its children, then second root item, etc.

The view context passed to the Items draw() method has the following properties:

* view - The view we're drawing to.
* cairo - The CairoContext to draw to.
* selected - True if the item is actually selected in the view.
* focused - True if the item has the focus.
* hovered - True if the mouse pointer is over the item. Only the top-most item is marked as hovered.
* dropzone - The item is marked as the drop zone. When this happens, then an item is dragged over the item, and if it is dropped it will become a child of this item.
* draw\_all - True if everything drawable on the item should be drawn, for example, when calculating the bounding boxes of an item.

The View automatically calculates the bounding box for the item based on the items drawn in the draw (context) function (this is only done when necessary, e.g., after an update of the item). The bounding box is in viewport coordinates.

The actual drawing is done by Painters (painter.py). A series of Painters have been defined—one for handles, one for items, etc.

### Tools

Behaviour is added to the Canvas (view) by tools. Tools can be chained together in order to provide more complex behaviour.

To make it easy, a DefaultTool has been defined which is a ToolChain instance with the tools added as follows:

* ToolChain - Delegates to a set of individual tools and keeps track of which tool has grabbed the focus. This normally happens when the user presses a mouse button. Once this happens, the tool requests a "grab" and events, like motion or button release, are sent directly to the focused tool.
* HoverTool - Makes the item under the mouse button the "hovered item." When such an item is drawn, its context.hovered\_item flag will be set to True.
* HandleTool - Allows for handles to be dragged around and focuses the item when its handle is clicked on.
* ItemTool - Selects items and enables dragging items around.
* TextEditTool - A demo tool that features a text edit pop-up.
* RubberbandTool - Invoked when the mouse button is pressed on a section of the view where no items or handles are present. It allows the user to select items using a "rubber band" selection box.

### Interaction

**Tools handle interaction with the Canvas view (visual component)**. Although the default tools do a fair amount of work, in most cases you’ll desire to create some custom connection **behavior (maintain consistent spelling)**. In order to implement these, HandleTool provides hooks including connect, disconnect, and glue.

One of the challenges you’ll likely face is what happens when an item is removed from the Canvas and there are other items (lines) connected to it. Gaphas provides a solution to this by providing a disconnect handler to the handle instance once it is connected. A function can be assigned to this disconnect handler, which is then called when the item that is connected to is removed from the Canvas.

### Undo

Gaphas has a simple built-in system for registering changes in its classes and notifying the application. This code resides in state.py.

There is also a "reverter" framework in place. The **system(framework?**) is notified when objects change their state, and the framework will figure out the reverse operation that has to be applied in order to undo the operation.

API

Consistency of Canvas capitalization

handle = Handle((10, 10), connectable=True)

No opening parenthesis

#### gaphas.painter.CairoBoundingBoxContext

It is used to intercept stroke(), fill(), and other context operations so that the bounding box of the item involved can be calculated.

Tools

* view (gaphas.view.View): The view to use for the tool, default is None.

This is repeated for all the classes. So maybe mention once and describe how it will be used in the other classes as well?

#### Class: gaphas.aspects.ElementHandleSelection

* view (gaphas.view.View): The view that can be used to apply the cursor to.

ConnectHandleTool (if a single tool) takes care of performing the user interaction required for this feature.

#### Class: gaphas.guide.GuidedItemInMotion

* view (gaphas.view.View): The view with guides to use for move coordinates.

 depth-first- I am assuming the user will know what this means.

Quadtrees are most often used to partition a two (sentence seems incomplete to me).

* capacity (int); The number of elements in one tree bucket; default is 10.

Schedule an ide(idle?) handler at a given priority.

* single (bool): Schedules the decorated function to be called only a single time.

#### Decorator: @Nonrecursive(to be consistent with the capitalization with the first one)

#### Decorator: @Recursive