qpageview Documentation

Release 0.6.2

Wilbert Berendsen

May 06, 2022

CONTENTS

qpageview provides a page based document viewer widget for Qt5/PyQt5.

It has a flexible architecture potentionally supporting many formats. Currently, it supports SVG documents, images, and, using the Poppler-Qt5 binding, PDF documents.

```
import qpageview

from PyQt5.Qt import *
a = QApplication([])

v = qpageview.View()
v.show()
v.loadPdf("path/to/afile.pdf")
```

Homepage • Development • Download • Documentation • License

CHAPTER

ONE

FEATURES

- Versatile View widget with many optional mixin classes to cater for anything between basic or powerful functionality
- Rendering in a background thread, with smart priority control, so display of large PDF documents remains fast and smooth
- Almost infinite zooming thanks to tile-based rendering and caching
- Magnifier glass
- Printing functionality, directly to cups or via Qt/QPrinter
- Can display pages originating from different documents at the same time
- Can show the difference between pages that are almost the same via color composition
- And much more! And…all classes are extendable and heavily customizable, so it is easy to inherit and add any functionality you want.

CHAPTER

TWO

DEPENDENCIES

- Python 3.6+
- Qt5
- PyQt5
- python-poppler-qt5 (needed for display of PDF documents)
- pycups (optionally, needed to print to a local CUPS server)

This manual documents *qpageview* version 0.6.2. Last update: May 06, 2022.

2.1 Basic usage

2.1.1 Creating the View widget

Just import *qpageview* and create a View. As the *View* is a QWidget, you need to create a QApplication object, just as for all Qt-based applications:

```
from PyQt5.QtWidgets import QApplication
import qpageview

app = QApplication([])

v = qpageview.View()
v.resize(900, 500)
v.show()
```

2.1.2 Loading contents

Load a PDF file with:

```
v.loadPdf("path/to/a_file.pdf")
```

or images, or SVG files:

```
import glob
v.loadImages(glob.glob("*.jpg"))
v.loadSvgs(glob.glob("*.svg"))
```

It is also possible to display pages originating from different sources at the same time in a View, see *Advanced usage*.

To clear the View again:

```
v.clear()
```

2.1.3 Navigating in the View

The View numbers pages starting from 1, like printed documents do. You can programmatically navigate through the View:

```
v.pageCount()               # get the number of pages
v.setCurrentPageNumber(11)  # go to page 11
v.currentPageNumber()       # get the current page number
v.gotoNextPage()            # go to the next page
v.gotoPreviousPage()        # go to the previous page
```

If the page you want to go to is not completely visible, it is scrolled into View.

2.1.4 Controlling the display

You can interact in the normal way with the widget, scrolling and zooming. Note the almost infinite zoom, thanks to the tile-based rendering engine.

There are various methods to change things, like *rotation*:

```
v.rotateRight()
v.rotateLeft()
v.setRotation(2)    # or v.setRotation(qpageview.Rotate_180)
```

or *zooming*:

```
v.zoomIn()
v.zoomOut()
v.setZoomFactor(2.0)
```

or *how* to fit the document while resizing the View widget:

```
v.setViewMode(qpageview.FitWidth)       # fits the page(s) in the width
v.setViewMode(qpageview.FitHeight)      # fits the page's height
v.setViewMode(qpageview.FitBoth)        # shows the full page
v.setViewMode(qpageview.FixedScale)     # don't adjust zoom to the widget
```

Setting the zoomFactor automatically switches to the FixedScale mode.

Change the *orientation*:

```
v.setOrientation(qpageview.Vertical)
v.setOrientation(qpageview.Horizontal)
```

Change the *continuous* mode:

```
v.setContinuousMode(False)      # only display the current page(s)
v.setContinuousMode(True)       # display all pages
```

Change the *layout mode*:

```
v.setPageLayoutMode("double_right") # Two pages, first page right
v.setPageLayoutMode("double_left")  # Two pages, first page left
v.setPageLayoutMode("single")       # Single pages
v.setPageLayoutMode("raster")       # Shows pages in a grid
```

(The method *pageLayoutModes()* returns a dictionary mapping the available layout mode names to the constructors of their corresponding layout engines. By making new *LayoutEngine* subclasses, you can implement more layout modes, and you can reimplement `pageLayoutModes()` to include them.)

All these properties have "getter" couterparts, like `viewMode()`, `orientation()`, etc.

2.1.5 The Magnifier

You can add a *Magnifier*:

```
from qpageview.magnifier import Magnifier
m = Magnifier()
v.setMagnifier(m)
```

Now, Ctrl+click in the View, and the Magnifier appears. You can also show the Magnifier programmatically with:

```
m.show()  # or v.magnifier().show()
```

Now you can only get it away with:

```
m.hide()
```

`Ctrl+Wheel` in the magnifier zooms the magnifier instead of the whole View. `Shift+Ctrl+Wheel` resizes the magnifier.

2.1.6 The Rubberband

You can add a *Rubberband*, to select a square range:

```
from qpageview.rubberband import Rubberband
r = Rubberband()
v.setRubberband(r)
```

By default with the right mousebutton you can select a region. The rubberband has various methods to access the selected area, just the rectangle, or the rectangle of every page the selection touches, or the selected square as an image or, depending on the underlying page type, the text or clickable links that fall in the selected region.

2.1.7 Controlling the behaviour

Scrolling

By default, the View has smooth and kinetic scrolling. Kinetic scrolling means that the View does not move the pages at once, but always scrolls with a decreasing speed to the desired location, which is easier on the eyes.

If you want to disable kinetic scrolling altogether, set the *kineticScrollingEnabled* attribute of the View to False.

If you only want to disable kinetic scrolling when paging through the document using the methods mentioned under *Navigating in the View*, you can leave *kineticScrollingEnabled* to True, but set *kineticPagingEnabled* to False.

Zooming

The user can zoom in and out with Ctrl+Mousewheel, which is expected behaviour. You can disable wheel zooming by setting the *wheelZoomingEnabled* attribute of View to False.

The minimum and maximum zoom factor can be set in the *MIN_ZOOM* and *MAX_ZOOM* attributes. By default you can zoom out to 5% and zoom in to 6400%.

Paging

By default, the `PageUp` and `PageDown` keys just scroll the View up or down ca. 90%. If you set the *strictPagingEnabled* attribute to True, in non-continuous mode those keys call the *gotoPreviousPage()* and *gotoNextPage()* methods, respectively.

2.2 Advanced usage

2.2.1 Document

A *View* displays *Page* objects, which optionally can belong to a *Document* object.

The convenience methods *View.loadPdf()*, *View.loadImages()* and *View.loadSvgs()*, create Document objects containing the pages, and then call *View.setDocument()* to display the pages in the view.

You can also use the module global functions like *loadPdf()* which return a Document, and then load that Document in the View:

```
v = qpageview.View()
v.show()

doc = qpageview.loadPdf("file.pdf")
v.setDocument(doc)
```

This way you can keep a document in memory, and you can load it, then load something else in the view and later load the same document again, without the need to load it again from disk or network.

When creating a Document using one of the global *load* functions, nothing is really loaded until you request the *pages()* of the Document, and even then, some Page types only load themselves really when their content is requested to be rendered in the View.

The list of individual Page objects in a document is returned by the *pages()* method of the Document class.

The current Page object (the current page number points to) is available through *View.currentPage()*.

2.2.2 Page and PageLayout

The View does not do very much with the Document it displays, rather it cares for the Page objects that are displayed.

The pages are in the PageLayout of the View, which inherits from the Python `list` type. Get the *PageLayout* of a View using *View.pageLayout()*. Using the regular `list` methods you can add or remove Page objects to the layout. Then you need to call *View.updatePageLayout()* to update the PageLayout, which will adjust size and position of the Pages.

Instead of the above, and maybe even better and easier, you can use the *modifyPages()* context manager of View, which will automatically update the layout when it exits:

```
with v.modifyPages() as pages:
    del pages[0]                # remove the first page
    pages.append(another_page)  # append another
```

This context manager yields the pages list, and when it exits it puts the pages in the layout, and updates the page layout. Note that in the layout, and in this `pages` list, the first page is at index 0.

This way, it is very easy to display Page objects originating from different sources:

```
import qpageview.image
page1 = qpageview.image.ImagePage.load("image.jpg")
page2 = qpageview.loadPdf("file.pdf").pages()[2]

with v.modifyPages() as pages:
    pages[:] = [page1, page2]       # [:] replaces the current contents
```

2.2.3 Controlling a view with ViewActions

Normally, in a Qt application, you create QActions to perform tasks and put those in a menu or toolbar. The *qpageview* package provides the *viewactions* module to help you with that.

If you create a *ViewActions* object and connect it to a View, all actions can readily be used to control the View, and they automatically update their state according to the View's state. The actions (QAction objects) are in the attributes of the ViewActions object.

For example, to add some actions to a menu:

```
import qpageview.viewactions
a = qpageview.viewactions.ViewActions()

a.setView(v)

menu = Qmenu()
menu.addAction(a.fit_width)
menu.addAction(a.fit_height)
menu.addAction(a.fit_both)
menu.addSeparator()
menu.addAction(a.zoom_in)
menu.addAction(a.zoom_out)

menu.popup(QCursor.pos())
```

The `pager` action fits well in a toolbar, it displays a spinbox where you can cycle through the pages, and the `zoomer` action displays a combobox with different zoom levels.

The full list of available action names is returned by the *names()* classmethod. You can set icons to the actions as you like, and replace the texts. It is also easy to inherit from ViewActions and add actions or change existing actions.

This is the list of actions that are currently available in a *ViewActions* object:

Name	Text	Action
print	Print	Open a print dialog
fit_width	Fit Width	Zoom to fit pages in the width of the View
fit_height	Fit Height	Zoom to fit pages in the height of the View
fit_both	Fit Both	Zoom to fit the full page in the View
zoom_natural	Natural Size	Zoom to a "natural" size (Page dpi/screen dpi)
zoom_original	Original Size	Set zoom factor to 1.0
zoom_in	Zoom in	
zoom_out	Zoom out	
zoomer	(none)	Display a *zoom widget* in a toolbar
rotate_left	Rotate Left	Rotate the pages 90° counter-clockwise
rotate_right	Rotate Right	Rotate the pages 90° clockwise
layout_single	Single Pages	Show single pages in a row
layout_double_right	Two Pages (first page right)	Show page 1 alone, to the right, then the rest two by two
layout_double_left	Two Pages (first page left)	Show pages two by two
layout_raster	Raster	Show pages in a grid
vertical	Vertical	Show the pages in a vertical row
horizontal	Horizontal	Show the pages in a horizontal row
continuous	Continuous	Checkbox, if checked shows all pages
reload	Reload	Reload pages from their files if possible
previous_page	Previous Page	Go to the previous page
next_page	Next Page	Go to the next page
pager	(none)	Display a *pager widget* in a toolbar
magnifier	Magnifier	Toggle the Magnifier visibility

Lazy View instantiation

It is possible to create a ViewActions object first and populate menus and toolbars with the actions, while the View is not yet created (e.g. when the View is in a dock widget that's only created when first shown). In this case, you want to instantiate the dock widget and View as soon as an action is triggered. To do this, connect to the `viewRequested()` signal of the ViewActions object. The connected method must create widgets as needed and then call *setView()* on the ViewActions object, so the action can be performed.

2.2.4 Managing View settings

All display settings (preferences) of a View can be stored in a QSettings object using *View.writeProperties()*, and read with *View.readProperties()*. These properties are: `position`, `rotation`, `zoomFactor`, `viewMode`, `orientation`, `continuousMode` and `pageLayoutMode`.

Under the hood, this is done using a *ViewProperties* object, which handles the saving and loading of properties, and getting/setting them from/to a View.

If you want the View to remember the position, zoom factor etc. on a per-document basis, you can install a *DocumentPropertyStore* in the View. This automatically stores the view properties for the current Document as soon as you load a different Document (using *View.setDocument()*). If you switch back to the former document, the View restores its position and other display settings for that document.

To use a DocumentPropertyStore:

```
v = qpageview.View()
store = qpageview.view.DocumentPropertyStore()
v.documentPropertyStore = store
```

By setting a mask it is possible to influence which properties are remembered. In this example, only zoom factor and position are remembered when switching documents:

```
store.mask = ['position', 'zoomFactor']
```

Lazy View instantiation: It is also possible to initialize the *ViewActions* from your settings, even if you have not yet created a View (for example, when the View is in a not yet created dock widget that is lazily instantiated). This way, you application's user interface already reflects the correct settings for the yet-not-created view. Use the View.properties() static method to get an uninitialized ViewProperties object, set some defaults and then add settings read from a QSettings object. Finally update the state of the actions in the ViewActions object, *before* connecting to the `ViewActions.viewRequested` signal.

All methods of ViewProperties return self, so these calls can be easily chained:

```
settings = QSettings()
props = qpageview.View.properties().setdefaults().load(settings)
actions = qpageview.viewactions.ViewActions()
actions.updateFromProperties(props)
actions.viewRequested.connect(createView)
```

Later, when you really instantiate the View, you should also load the View settings; the ViewActions object does not actively update the View when connecting (rather, the actions are adjusted to the View when connecting):

```
def createView():
    # creating the View....
    v = qpageview.View()
    settings = QSettings()
```

(continues on next page)

(continued from previous page)

```
    v.readProperties(settings)
    actions.setView(v)
```

2.2.5 Using View Mixins

The View as defined in the *qpageview* module is a class composed of the basic View class in *view.View* and some View Mixin classes that extend the functionality of the basic View.

This is a list of the currently available View Mixin classes:

link.LinkViewMixin Adds functionality to click on links, e.g. in PDF pages

highlight.HighlightViewMixin Adds functionality to highlight rectangular regions

shadow.ShadowViewMixin Draws a nice shadow border around the pages

util.LongMousePressMixin Handles long mouse presses (can be mixed in with any QWidget)

imageview.ImageViewMixin A View targeted to the display of one single image (see also the *ImageView*)

selector.SelectorViewMixin Adds functionality to make pages selectable with a checkbox

widgetoverlay.WidgetOverlayViewMixin Adds functionality to display QWidgets on Pages that scroll and optionally zoom along and the user can interact with

So, depending on your needs, you can create your own View subclass, mixing in only the functionality you need. Put the main View class at the end, for example:

```
class View(
    qpageview.link.LinkViewMixin,
    # other mixins here
    qpageview.view.View):
    """My View with some enhancements."""
    pass

    # my own extensions and new funcionality
    def myMethod(self):
        pass
```

2.2.6 Specialized View subclasses

There are already some specialized View subclasses available, those are:

ImageView A View that is tailored to show one image (from file, data or a QImage)

SidebarView A View that shows selectable thumbnails of all pages in a connected View, usable as a sidebar for a normal View.

2.3 Interacting with pages

2.3.1 Coordinate systems

A Page can display text or graphics, have clickable links, etc. A Page also has certain dimensions and its own notion of natural size, via the `dpi` attribute of the Page (sub)class.

There are three ways of determining a position on a Page:

1. The pixel position on the Page in a View, e.g. where a mouse button is pressed. A Page knows its current dimensions in pixels: in the Page's `width` and `height` instance attributes, and as a QSize via the `size()` method. If a Page is rotated 90 or 270 degrees, then the Page's original height now corresponds to the displayed page's width in pixels.

   In most cases this is called "page coordinates." Page coordinates are always integer values.

2. A position on the Page in its default size and without rotation. The original size of a Page is independent of the current zoomFactor of the View, and rather determined by the underlying image, SVG or PDF file. This is used e.g. when printing or converting to vector formats. The original size is accessible via the `pageWidth` and `pageHeight` attributes, and as a QSizeF via the *pageSize()* method.

   This is called "original page coordinates." Normally these are floating point values.

   (When the `dpi` Page class attribute is the same as the current DPI setting of the computer's display, then the displayed size of a Page at zoom factor 1.0 in pixels is the same as the default size.)

3. A position where both horizontal and vertical offset are floating point, in the range 0..1, without rotation. This is used to determine the position of links, rectangular areas to highlight, and to position overlay widgets by the widget overlay view mixin.

Page has the method *transform()* to get a QTransform matrix that can map between page coordinates and original or 0..1 coordinates. The methods *mapToPage()* and *mapFromPage()* return helper objects that can convert QPoints and QRects from and to original page coordinates. These matrices take into account the page's scaling and current rotation, and they always return floating point values for original or 0..1 range coordinates, and integers for page coordinates.

2.3.2 Page position and Layout position

Many methods neatly hide the computations between mouse cursor position and position in original page coordinates on a particular page, but it is still nice to understand it a bit.

A PageLayout is just a large rectangular (virtual) area, large enough so that all Pages in the layout can be set to a position and size so that they do not overlap. Every Page is assigned a `pos()` on the layout. The geometry() of the layout is the rectangle encompassing all visible pages on the layout.

View.layoutPosition() returns the position of the layout relative to the top-left corner of the View's viewport. You can find the pages that are currently visible using *View.visiblePages()*. To find the Page the mouse cursor points at, use:

```
# pos is mouse position in viewport
pos_on_layout = pos - view.layoutPosition()
page = view.pageLayout().pageAt(pos)
pos_on_page = pos_on_layout - page.pos()

# translate the pixel position to original page coordinates
pos = page.mapFromPage().point(pos_on_page)
```

2.3.3 Links on a page

A Page can contain clickable links, which are collected in a Links object that is available under the *links()* method of Page.

Every *Link* has at least an `url` property and an `area` property, which contains the rectangle of the clickable area in four coordinates in the 0..1 range.

You could use the above logic to access links on the page, but if you use the LinkViewMixin class in your View class, there are simple methods: For example, *View.linkAt()* returns the link at the specified mouse cursor position, if any. To get an understanding of how things work under the hood is here the implementation of that method:

```
class View:
    # (...)
    def linkAt(self, pos):
        """If the pos (in the viewport) is over a link, return a (page, link) tuple.

        Otherwise returns (None, None).

        """
        pos = pos - self.layoutPosition()
        page = self.pageLayout().pageAt(pos)
        if page:
            links = page.linksAt(pos - page.pos())
            if links:
                return page, links[0]
        return None, None
```

We see that first the mouse cursor position is translated to the layout's position, and then the layout is asked for a page on that position (*PageLayout.pageAt()*). If a page is there, the position is translated to the page: `pos - page.pos()` (coordinates (0, 0) is the top-left corner of the Page).

Then the page is asked for links at that position. Let's look at the implementation of *Page.linksAt()*:

```
class Page:
    # (...)
    def linksAt(self, point):
        """Return a list() of zero or more links touched by QPoint point.

        The point is in page coordinates.
        The list is sorted with the smallest rectangle first.

        """
        # Link objects have their area ranging
        # in width and height from 0.0 to 1.0 ...
        pos = self.mapFromPage(1, 1).point(point)
        links = self.links()
        return sorted(links.at(pos.x(), pos.y()), key=links.width)
```

We see that a matrix is used to map from page pixel coordinates to original coordinates, but in the 0..1 range. Then the Links object is queried for links at that position, sorted on width. The smallest one at that position is ultimately returned by *View.linkAt()*.

Both PageLayout and Links internally use *rectangles.Rectangles* to manage possibly large groups of rectangular objects and quickly find intersections with those objects and a point or rectangle.

Links in a Document

All links in a Document can be requested with *Document.urls()*. This method returns a dictionary where the url is the key, and the value is a dictionary mapping page number to a list of rectangular areas of all links with that url on that page.

2.3.4 Getting text from a page

Besides links, depending on the Page type, a page can also contain text, such as PDF pages do. You can get the text with the *Page.text()* method, which returns the text in a rectangle in page coordinates:

```
page = view.currentPage()

# get the text in some rectangle
text = page.text(some_rect)

# get the full text by using the page's rectangle
full_text = page.text(page.rect())

# using the rubberband selection
text = view.rubberband().selectedText()
```

2.3.5 Getting image data from a page

You can get pixel data using *Page.image()*:

```
image = page.image()
```

This method returns a QImage. See the documentation for the arguments to this function, to adjust the resolution and the area (which defaults to the whole page).

You can also get graphic data in *PDF*, *EPS* or *SVG* format. For document formats that are vector based, this graphic data wil also be vector based. For example:

```
page.pdf("filename.pdf")
page.svg("filename.svg")
page.eps("filename.eps")

# using the rubberband selection:
page, rect = view.rubberband.selectedPage()
if page:
    page.pdf("filename.pdf", rect)
```

See the method's documentation for more information about possible arguments to these functions. Instead of a filename, you can also give a QIODevice object. All these functions return True if they were successful.

For more advanced methods to get image data, see the *export* module.

2.4 How rendering works

To render Page objects graphically, a Page class should implement three methods: *paint()*, *print()* and *image()*.

- `paint()` is used to paint the image in the View, in page coordinates. If painting is expensive, this method should return immediately and schedule a pixmap to be drawn in a background thread (see below).
- `print()` is used to paint the image to a QPainter on any QPaintDevice, in original coordinates (i.e. the used QPainter has already been transformed to the original page size without rotation).
- `image()` is used to get a rendered QImage.

Most Page classes depend on a *Renderer* that implements the actual rendering. The base Renderer class has functionality for caching and for tile-based rendering in a background thread, so when you zoom in very far, only a small portion of the original page is drawn on a pixmap to be displayed on the screen.

Awaiting the rendering, the View scales another image from the cache of the same region (if available) to display instead.

It is not necessary to specify a renderer directly, although it can be useful. All builtin page classes install a default renderer. Page types that use a renderer inherit from *page.AbstractRenderedPage*.

2.4.1 Available Page types

These are the currently available Page types, and their corresponding Document types:

Module	Page type	Document type	Displays
image	*ImagePage*	*ImageDocument*	all image formats supported by QImage
svg	*SvgPage*	*SvgDocument*	SVG images, one file per page
poppler	*PopplerPage*	*PopplerDocument*	PDF documents, multiple pages per file
diff	*DiffPage*	*DiffDocument*	color composites other pages of any type

2.4.2 Implementing a new page type

If you study the source of the *svg* module, you can see that there is only very little code needed to implement a rendered Page type.

For the rendered Page, *Page.paint()* calls *Renderer.paint()*, which schedules an image to be generated. The image is generated by *Renderer.render()*, which by default calls *Renderer.draw()*, which does the actual drawing work. Also *Page.print()* calls *Renderer.draw()* directly, while *Page.image()* simply calls *Renderer.image()*, which also calls *Renderer.render()*, which in turns calls *Renderer.draw()*.

So you actually only need to implement *Renderer.draw()* :-) But, depending on the characteristics of the underlying graphics type, other strategies may be combined to achieve a well-working Page type.

2.5 Overview of all modules

2.5.1 The main qpageview module

This is a generic paged view widget.

Its main design goal is to display the pages of a PDF document, but it can display any set of images or pages, originating from different documents.

Every page is represented by a Page instance, which encompasses all logic for the document type, i.e. drawing etc.

Pages are managed by a PageLayout.

A PageLayout can be set to a View so the pages are displayed.

The images from a PDF, SVG of possibly other document are cached, and rendering is tile-based, to support zooming in at great detail. Also a magnifier is available, which by default pops up at Ctrl+click.

Because the qpageview is built on Qt, we use the Qt convention to have camelCase method names and CamelCase class names.

class View(*parent=None*, ***kwds*)

Bases: *qpageview.link.LinkViewMixin*, *qpageview.highlight.HighlightViewMixin*, *qpageview.shadow.ShadowViewMixin*, *qpageview.util.LongMousePressMixin*, *qpageview.view.View*

Paged view component based on view.View, with all enhancements.

loadPdf(*filename*, *renderer=None*)

Convenience function to create a Document with the specified PDF file.

The filename can also be a QByteArray or an already loaded popplerqt5.Poppler.Document instance.

loadSvgs(*filenames*, *renderer=None*)

Convenience function to create a Document with the specified SVG files.

Each SVG file is loaded in one Page. A filename can also be a QByteArray.

loadImages(*filenames*, *renderer=None*)

Convenience function to create a Document with images from files.

Each image is loaded in one Page. A filename can also be a QByteArray.

diffDocument(*documents*, *renderer=None*)

Convenience function to create a diff.DiffDocument from other documents.

The difference works best with documents that are similar and mostly black and white.

2.5.2 The backgroundjob module

Run jobs in the background using QThread.

class Job(*parent=None*)

Bases: `PyQt5.QtCore.QThread`

A simple wrapper around QThread.

Before calling start() you should put the work function in the work attribute, and an optional finalize function (which will be called with the result) in the finalize attribute.

Or alternatively, inherit from this class and implement work() and finish() yourself. The result of the work function is stored in the result attribute.

finalize = None

running = False

done = False

result = None

start(*self*, *priority: QThread.Priority = QThread.InheritPriority*)

run()

Call the work function in the background thread.

work()

Implement this to get the work done.

If you have long tasks you can Qt's isInterruptionRequested() functionality.

Instead of implementing this method, you can put the work function in the work instance attribute.

finish()

This slot is called in the main thread when the work is done.

The default implementation calls the finalize function with the result.

class SingleRun

Bases: object

Run a function in a background thread.

The outcome is silently discarded if another function is called before the old one finishes.

cancel()

Forgets the running job.

The job is not terminated but the callback is not called.

run(*func*, *callback=None*)

Run specified function in a background thread.

The thread is immediately started. If a callback is specified, it is called in the main thread with the result when the function is ready.

2.5.3 The cache module

Cache logic.

class ImageEntry(*image*)

Bases: object

class ImageCache

Bases: object

Cache generated images.

Store and retrieve them under a key (see render.Renderer.key()).

maxsize = 209715200

currentsize = 0

clear()

Remove all cached images.

invalidate(*page*)

Clear cache contents for the specified page.

tileset(*key*)

Return a dictionary with tile-entry pairs for the key.

If no single tile is available, an empty dict is returned.

addtile(*key*, *tile*, *image*)

Add image for the specified key and tile.

closest(*key*)

Iterate over suitable image tilesets but with a different size.

Yields (width, height, tileset) tuples.

This can be used for interim display while the real image is being rendered.

2.5.4 The constants module

Constant values.

Rotate_0 = 0

normal

Rotate_90 = 1

90° rotated clockwise

Rotate_180 = 2

180° rotated

Rotate_270 = 3

270° rotated (90° couter clockwise)

FixedScale = 0

the scale is not adjusted to the widget size

FitWidth = 1

scale so that the page's width fits in the widget

FitHeight = 2

scale so that the page's height fits in the widget

FitBoth = 3

fit the whole page

Horizontal = 1

arrange the pages in horizontal order

Vertical = 2

arrange the pages in vertical order

2.5.5 The cupsprinter module

A simple module using CUPS to send a document directly to a printer described by a QPrinter. This is especially useful with PDF documents.

Uses the *cups* module, although it elegantly fails when that module is not present. The cups module can be found in the pycups package at https://pypi.org/project/pycups/ .

There are two methods to send a document to a printer:

1. Using the *lp* shell command
2. Using the cups module, which uses libcups to directly contact the server.

This module provides both possibilities.

Use *CmdHandle.create()* to get a CmdHandle, if *lp* is available, or use *IppHandle.create()* to get a IppHandle, if the cups module is available and a connection to the server can be established.

A function *handle()* is available; that tries first to get an IppHandle and then a LprHandle. Usage of this module is this simple:

```
import qpageview.cupsprinter

h = qpageview.cupsprinter.handle()
if h:
    h.printFile('/path/to/document.pdf')
```

You can supply a QPrinter instance (that'd be the normal workflow :-)

```
h = qpageview.cupsprinter.handle(printer)
if h:
    h.printFile('/path/to/document.pdf')
```

In this case all options that are set in the QPrinter object will be used when sending the document to the printer.

If *printFile()* returns True, printing is considered successful. If False, you can read the *status* and *error* attributes:

```
if not h.printFile('/path/to/document.pdf'):
    QMessageBox.warning(None, "Printing failure",
        "There was an error:\n{0} (status: {1})".format(h.error, h.status))
```

To print a list of files in one job, use *printFiles()*.

class Handle(*printer=None*)

Bases: object

Shared implementation of a handle that can send documents to a printer.

setPrinter(*printer*)

Use the specified QPrinter.

printer()

Return the QPrinter given on init, or a new default QPrinter instance.

options()

Return the dict of CUPS options read from the printer object.

title(*filenames*)

Return a sensible job title based on the list of filenames.

This method is called when the user did not specify a job title.

printFile(*filename*, *title=None*, *options=None*)

Print the file.

printFiles(*filenames*, *title=None*, *options=None*)

Print a list of files.

If the title is None, the basename of the filename is used. Options may be a dictionary of CUPS options. All keys and values should be strings.

Returns True if the operation was successful. Returns False if there was an error; after the call to printFile(), the status and error attributes contain the returncode of the operation and the error message.

class CmdHandle(*command*, *server=''*, *port=0*, *user=''*, *printer=None*)

Bases: *qpageview.cupsprinter.Handle*

Print a document using the *lp* shell command.

classmethod create(*printer=None*, *server=''*, *port=0*, *user=''*, *cmd='lp'*)

Create a handle to print using a shell command, if available.

class IppHandle(*connection*, *printer=None*)

Bases: *qpageview.cupsprinter.Handle*

Print a document using a connection to the CUPS server.

classmethod create(*printer=None*, *server=''*, *port=0*, *user=''*)

Return a handle to print using a connection to the (local) CUPS server, if available.

handle(*printer=None*, *server=''*, *port=0*, *user=''*)

Return the first available handle to print a document to a CUPS server.

options(*printer*)

Return the dict of CUPS options read from the QPrinter object.

clearPageSetSetting(*printer*)

Remove 'page-set' even/odd cups options from the printer's CUPS options.

Qt's QPrintDialog fails to reset the 'page-set' option back to 'all pages', so a previous value (even or odd) could remain in the print options, even if the user has selected All Pages in the print dialog.

This function clears the page-set setting from the cups options. If the user selects or has selected even or odd pages, it will be added again by the dialog.

So call this function on a QPrinter, just before showing a QPrintDialog.

2.5.6 The diff module

A Page intended to display the visual difference between other pages.

class DiffPage(*renderer=None*)

Bases: *qpageview.page.ImagePrintPageMixin*, *qpageview.multipage.MultiPage*

A Page that shows the difference between sub pages.

DiffPage inherits from MultiPage; the pages are to be added in the pages attribute. The first page is considered to be the "default" page, shown the normal way; the others are added in configurable colors and intensity.

opaquePages = False

classmethod createPages(*pageLists*, *renderer=None*, *pad=<class 'qpageview.page.BlankPage'>*)

Reimplemented to adapt the page sizes.

renderer = <qpageview.diff.DiffRenderer object>

class DiffDocument(*sources=()*, *renderer=None*)

Bases: *qpageview.multipage.MultiPageDocument*

A Document showing the differences between documents, set as sources.

pageClass

alias of *qpageview.diff.DiffPage*

class DiffRenderer

Bases: *qpageview.multipage.MultiPageRenderer*

Renders the pages by calling their own renderer.

How the difference is displayed can be configured using this renderer. Up to four different pages can be displayed, the colors to render them are taken from the colors instance variable, which is a list.

The alpha channel of each color determines the visiblity of the corresponding sub page.

This renderer works best with pages that are mostly black on a white background.

combine(*painter*, *images*)

Paint images on the painter.

We draw bottom-up, using Darken composition mode, so the lower images remain visible.

2.5.7 The document module

Document, a simple class representing a group of pages.

It is certainly not necessary to use a Document to handle pages in a View, but it might be convenient in some cases.

The Document class can be used to manually build a document consisting of a group of pages, that can be specified on construction or added to the list returned by the Document.pages() method.

Then two subtypes exist, SingleSourceDocument and MultiSourceDocument, that can be subclassed into document types that either load every Page from a single file or, respectively, load all pages from one filename.

Instead of a filename, any object can be used as data source. Depending on the page type, a QIODevice or QByteArray could be used.

Instantiating a Document is very fast, as nothing is loaded or computed on instantiation. Only when pages() is called for the first time, file contents are loaded, which normally happens when a Document is shown in a View using View.setDocument().

class Document(*pages=()*)

Bases: `object`

A Document represents a group of pages that belong together in some way.

Add pages on creation or by manipulating the list returned by pages().

count()

Return the number of pages.

pages()

Return the list of pages.

clear()

Empties the document.

filename()

Return the filename of the document.

The default implementation returns an empty string.

filenames()

Return the list of filenames, for multi-file documents.

The default implementation returns an empty list.

urls()

Return a dict, mapping URLs (str) to areas on pages.

This method queries the links of all pages, and if they have a URL, the area attribute of that link is added to a list for every page, and every unique URL is mapped to a dict, that maps page number to the list of areas on that page (page numbers start with 0).

In the returned dict you can quickly find the areas in which a URL appears in a link.

addUrls(*urls*)

Read the dict (such as returned by urls()) and make clickable links.

This can be used to add url-links to a document from another document, e.g. when a document represents the same content, but has no clickable links (e.g. images). Links on pages with a higher number than our number of pages are skipped.

class AbstractSourceDocument(*renderer=None*)

Bases: *qpageview.document.Document*

A Document that loads pages from external source, such as a file.

The pages are loaded on first request, and invalidate can be called to trigger a reload.

pages()

Return the list of Pages, creating them at first call.

invalidate()

Delete all cached pages, except for filename(s) or source object(s).

Also called internally by clear().

clear()

Delete all cached pages, and clear filename(s) or source object(s).

createPages()

Implement this method to create and yield the pages.

This method is only called once. After altering filename,-s or source,-s, or invalidate(), it is called again.

urls()

Reimplemented to cache the urls returned by Document.urls().

class SingleSourceDocument(*source=None*, *renderer=None*)

Bases: *qpageview.document.AbstractSourceDocument*

A Document that loads its pages from a single file or source.

source()

Return a data object that might be set for the whole document.

setSource(*source*)

Set the data object for the whole document. Invalidates the document.

filename()

Return the file name applying to the whole document.

setFilename(*source*)

Set the data object for the whole document. Invalidates the document.

clear()

Delete all cached pages, and clear filename or source object.

class MultiSourceDocument(*sources=()*, *renderer=None*)

Bases: *qpageview.document.AbstractSourceDocument*

A Document that loads every page from its own file or source.

sources()

Return data objects for every page.

setSources(*sources*)

Set data objects for every page. Invalidates the document.

filenames()

Return the list of file names of every page.

setFilenames(*sources*)

Set data objects for every page. Invalidates the document.

clear()

Delete all cached pages, and clear filenames or source objects.

2.5.8 The export module

Export Pages to different file formats.

class AbstractExporter(*page*, *rect=None*)

Bases: object

Base class to export a rectangular area of a Page to a file.

Specialized subclasses implement each format.

You instantiate a subclass with a Page and a rectangle. The rectangle may be None, to specify the full page. After instantiation, you can set attributes to configure the export. The following attributes are supported:

```
resolution = 300
autocrop = False
oversample = 1
grayscale = False
paperColor = None
forceVector = True  # force the render backend to be Arthur for
                    # exporting PDF pages to vector-based formats
```

After setting the attributes, you call one or more of save(), copyData(), copyFile(), mimeData() or tempFileMimeData(), which will trigger the export because they internally call data(), which caches its return value until setPage() is called again.

Not all exporters support all attributes, the supportXXX attributes specify whether an attribute is supported or not.

resolution = 300

antialiasing = True

autocrop = False

oversample = 1

grayscale = False

paperColor = None

forceVector = True

wantsVector = True

supportsResolution = True

supportsAntialiasing = True

supportsAutocrop = True

supportsOversample = True

supportsGrayscale = True

supportsPaperColor = True

mimeType = 'application/octet-stream'

filename = ''

defaultBasename = 'document'

defaultExt = ''

setPage(*page*, *rect=None*)

page()

Return our page, setting the renderer to our preferences.

autoCroppedRect()

Return the rect, autocropped if desired.

export()

Perform the export, based on the settings, and return the exported data object.

successful()

Return True when export was successful.

data()

Return the export result, assuming it is binary data of the exported file.

document()

Return a one-page Document to display the image to export.

Internally calls createDocument(), and caches the result, setting the papercolor to the papercolor attribute if the exporter supports papercolor.

createDocument()

Create and return a one-page Document to display the image to export.

renderer()

Return a renderer for the document(). By default, None is returned.

copyData()

Copy the QMimeData() to the clipboard.

mimeData()

Return a QMimeData() object representing the exported data.

save(*filename*)

Save the exported image to a file.

suggestedFilename()

Return a suggested file name for the file to export.

The name is based on the filename (if set) and also contains the directory path. But the name will never be the same as the filename set in the filename attribute.

tempFilename()

Save data() to a tempfile and returns the filename.

tempFileMimeData()

Save the exported image to a temp file and return a QMimeData object for the url.

copyFile()

Save the exported image to a temp file and copy its name to the clipboard.

pixmap(*size=100*)

Return a small pixmap to use for dragging etc.

drag(*parent*, *mimeData*)

Called by dragFile and dragData. Execs a QDrag on the mime data.

dragData(*parent*)

Start dragging the data. Parent can be any QObject.

dragFile(*parent*)

Start dragging the data. Parent can be any QObject.

class ImageExporter(*page*, *rect=None*)

Bases: *qpageview.export.AbstractExporter*

Export a rectangular area of a Page (or the whole page) to an image.

wantsVector = False

defaultBasename = 'image'

defaultExt = '.png'

export()
Create the QImage representing the exported image.

image()

createDocument()
Create and return a one-page Document to display the image to export.

copyData()
Copy the QMimeData() to the clipboard.

mimeData()
Return a QMimeData() object representing the exported data.

save(*filename*)
Save the exported image to a file.

class SvgExporter(*page*, *rect=None*)
Bases: *qpageview.export.AbstractExporter*

Export a rectangular area of a Page (or the whole page) to a SVG file.

mimeType = 'image/svg'

supportsGrayscale = False

supportsOversample = False

defaultBasename = 'image'

defaultExt = '.svg'

export()
Perform the export, based on the settings, and return the exported data object.

createDocument()
Create and return a one-page Document to display the image to export.

class PdfExporter(*page*, *rect=None*)
Bases: *qpageview.export.AbstractExporter*

Export a rectangular area of a Page (or the whole page) to a PDF file.

mimeType = 'application/pdf'

supportsGrayscale = False

supportsOversample = False

defaultExt = '.pdf'

export()
Perform the export, based on the settings, and return the exported data object.

createDocument()
Create and return a one-page Document to display the image to export.

class EpsExporter(*page*, *rect=None*)
Bases: *qpageview.export.AbstractExporter*

Export a rectangular area of a Page (or the whole page) to an EPS file.

mimeType = 'application/postscript'

supportsGrayscale = False

supportsOversample = False

defaultExt = '.eps'

export()

Perform the export, based on the settings, and return the exported data object.

createDocument()

Create and return a one-page Document to display the image to export.

pdf(*filename*, *pageList*, *resolution=72*, *paperColor=None*)

Export the pages in pageList to a PDF document.

filename can be a string or any QIODevice. The pageList is a list of the Page objects to export.

Normally vector graphics are rendered, but in cases where that is not possible, the resolution will be used to determine the DPI for the generated rendering.

The computedRotation attribute of the pages is used to determine the rotation.

Make copies of the pages if you run this function in a background thread.

2.5.9 The highlight module

Highlight rectangular areas inside a View.

class Highlighter

Bases: object

A Highlighter can draw rectangles to highlight e.g. links in a View.

An instance represents a certain type of highlighting, e.g. of a particular style. The paintRects() method is called with a list of rectangles that need to be drawn.

To implement different highlighting behaviour just inherit paintRects(). The default implementation of paintRects() uses the *color* attribute to get the color to use and the *lineWidth* (default: 2) and *radius* (default: 3) attributes.

lineWidth specifies the thickness in pixels of the border drawn, *radius* specifies the distance in pixels the border is drawn (by default with rounded corners) around the area to be highlighted. *color* is set to None by default, causing the paintRects method to choose the application's palette highlight color.

lineWidth = 2

radius = 3

color = None

paintRects(*painter*, *rects*)

Override this method to implement different drawing behaviour.

class HighlightViewMixin(*parent=None*, ***kwds*)

Bases: object

Mixin methods vor view.View for highlighting areas.

This mixin allows for highlighting rectangular areas on pages. You can highlight different sets of areas independently, using different Highlighter instances.

Highlighting can be set to stay on forever or to disappear after a certain amount of microseconds.

If desired, the View can be scrolled to show the highlighted areas. How the highlighting is drawn is determined by the paintRects() method of Highlighter.

defaultHighlighter()

Return a default highlighter, creating it if necessary.

setDefaultHighlighter(*highlighter*)

Set a Highlighter to use as the default highlighter.

highlightRect(*areas*)

Return the bounding rect of the areas.

highlight(*areas*, *highlighter=None*, *msec=0*, *scroll=False*, *margins=None*, *allowKinetic=True*)

Highlight the areas dict using the given or default highlighter.

The areas dict maps Page objects to lists of rectangles, where the rectangle is a QRectF() inside (0, 0, 1, 1) like the area attribute of a Link.

If the highlighter is not specified, the default highlighter will be used.

If msec > 0, the highlighting will vanish after that many microseconds.

If scroll is True, the View will be scrolled to show the areas to highlight if needed, using View.ensureVisible(highlightRect(areas), margins, allowKinetic).

clearHighlight(*highlighter=None*)

Removes the highlighted areas of the given or default highlighter.

isHighlighting(*highlighter=None*)

Return True if the given or default highlighter is active.

highlightUrls(*urls*, *highlighter=None*, *msec=0*, *scroll=False*, *margins=None*, *allowKinetic=True*)

Convenience method highlighting the specified urls in the Document.

The urls argument is a list of urls (str); the other arguments are used for calling highlight() on the areas returned by getUrlHighlightAreas(urls).

getUrlHighlightAreas(*urls*)

Return the areas to highlight all occurrences of the specified URLs.

The areas are found in the dictionary returned by document().urls(). URLs that are not in that dictionary are silently skipped. If there is no document set this method returns nothing.

paintEvent(*ev*)

Paint the highlighted areas in the viewport.

2.5.10 The image module

A page that can display an image, loaded using QImage.

ImagePages are instantiated quite fast. The image is only really loaded on first display.

class ImageContainer(*image*)

Bases: `object`

Represent an image, is shared among copies of the "same" Page.

size()

image(*clip=None*)

class ImageLoader(*source*, *autoTransform=True*)

Bases: *qpageview.image.ImageContainer*

Represent an image loaded from a file or IO device.

size()

Return the size of the image.

If the image can't be loaded, a null size is returned. The resulting value is cached.

image(*clip=None*)

Load and return the image.

If clip is given, it should be a QRect describing the area to load.

class ImagePage(*container*, *renderer=None*)

Bases: *qpageview.page.AbstractRenderedPage*

A Page that displays an image in any file format supported by Qt.

autoTransform = True

dpi = 96

classmethod load(*filename*, *renderer=None*)

Load the image and yield one ImagePage instance if loading was successful.

classmethod fromImage(*image*, *renderer=None*)

Instantiate one ImagePage from the supplied QImage.

As the image is kept in memory, it is not advised to instantiate many Page instances this way. Use load() for images on the filesystem. The image must be valid, and have a size > 0.

print(*painter*, *rect=None*, *paperColor=None*)

Paint a page for printing.

image(*rect=None*, *dpiX=None*, *dpiY=None*, *paperColor=None*)

Return a QImage of the specified rectangle.

group()

Return the group the page belongs to.

This could be some document structure, so that different Page objects could refer to the same graphical contents, preventing double caching.

This object is used together with the value returned by ident() as a key to cache the page. The idea is that the contents of the page are uniquely identified by the objects returned by group() and ident().

This way, when the same document is opened in multiple page instances, only one copy resides in the (global) cache.

By default, the page object itself is returned.

mutex()

Return an object that should be locked when rendering the page.

Page are guaranteed not to be rendered at the same time when they return the same mutex object. By default, None is returned.

renderer = <qpageview.image.ImageRenderer object>

class ImageDocument(*sources=()*, *renderer=None*)

Bases: *qpageview.document.MultiSourceDocument*

A Document representing a group of images.

A source may be a filename, a QIODevice or a QImage.

pageClass

alias of *qpageview.image.ImagePage*

createPages()

Implement this method to create and yield the pages.

This method is only called once. After altering filename,-s or source,-s, or invalidate(), it is called again.

class ImageRenderer(*cache=None*)

Bases: *qpageview.render.AbstractRenderer*

draw(*page*, *painter*, *key*, *tile*, *paperColor=None*)

Draw the specified tile of the page (coordinates in key) on painter.

2.5.11 The imageview module

ImageView, a View optimized for display of one Page, e.g. one image.

Clicking in the view toggles between FitBoth and NaturalSize.

class ImageViewMixin(*parent=None*)

Bases: `object`

View Mixin with a few customisations for displaying a single page/image.

Adds the instance variable:

fitNaturalSizeEnabled = True

If True, the image will not be scaled larger than its natural size when FitWidth, -Height, or -Both is active.

fitNaturalSizeEnabled = True

setImage(*image*)

Convenience method to display a QImage.

toggleZooming()

Toggles between FitBoth and natural size.

fitPageLayout()

Reimplemented to avoid zooming-to-fit larger than naturalsize.

mouseReleaseEvent(*ev*)

Reimplemented to toggle between FitBoth and ZoomNaturalSize.

class ImageView(*parent=None*)

Bases: *qpageview.imageview.ImageViewMixin*, *qpageview.view.View*

A View, optimized for display of one Page, e.g. one image.

Append one Page to the layout, use one of the load* methods to load a single page document, or use the setImage() method to display a QImage.

clickToSetCurrentPageEnabled = False

whether a mouse click in a page makes it the current page

2.5.12 The layout module

Manages and positions a group of Page instances.

class PageRects(*objects=None*)

Bases: *qpageview.rectangles.Rectangles*

get_coords(*page*)

You should implement this method.

The result should be a four-tuple with the coordinates of the rectangle the object represents (x, y, x2, y2). These are requested only once. x should be < x2 and y should be < y2.

class PageLayout(*iterable=(), /*)

Bases: *qpageview.util.Rectangular*, `list`

Manages page.Page instances with a list-like api.

You can iterate over the layout itself, which yields all Page instances.

The following instance attributes are used, with these class-level defaults:

```
zoomFactor = 1.0
dpiX = 72.0
dpiY = 72.0
rotation = Rotate_0
orientation = Vertical
alignment = Qt.AlignCenter
```

The layout has margins around each page, accessible via pageMargins(), and margins around the whole layout, accessible via margins(). Both have class level defaults as a tuple, but they are converted to a QMargins object for the layout instance when first accessed via the margins() and pageMargins() methods:

```
_margins = (6, 6, 6, 6)
_pageMargins = (0, 0, 0, 0)

spacing = 8             # pixels between pages

x = 0                   # x, y, width and height are set by update()
y = 0
```

(continues on next page)

(continued from previous page)

```
width = 0
height = 0

continuousMode = True   # whether to show all pages
```

The actual layout is done by a LayoutEngine in the engine attribute. After having changed pages, engine or layout attributes, call update() to update the layout.

spacing = 8

zoomFactor = 1.0

dpiX = 72.0

dpiY = 72.0

rotation = 0

orientation = 2

alignment = 132

continuousMode = True

currentPageSet = 0

count()
Return the number of Page instances.

empty()
Return True if there are zero pages.

setMargins(*margins*)
Sets our margins to a QMargins object.

margins()
Return our margins as a QMargins object, intialized from _margins

setPageMargins(*margins*)
Sets our page margins to a QMargins object.

pageMargins()
Return our page margins as a QMargins object, intialized from _pageMargins

pageAt(*point*)
Return the page that contains the given QPoint.

If the point is not on any page, None is returned.

pagesAt(*rect*)
Yield the pages touched by the given QRect.

The pages are in undefined order.

nearestPageAt(*point*)
Return the page at the shortest distance from the given point.

The returned page does not contain the point. (Use pageAt() for that.) If there are no pages outside the point, None is returned.

defaultWidth(*page*)

Return the default width of the page.

defaultHeight(*page*)

Return the default height of the page.

widestPage()

Return the page with the largest default width, if any.

highestPage()

Return the page with the largest default height, if any.

fit(*size*, *mode*)

Fits the layout in the given size (QSize) and ViewMode.

zoomsToFit()

Return True if the layout engine changes the zoomFactor to fit.

update()

Compute the size of all pages and updates their positions. Finally set our own size.

You should call this after having added or deleted pages or after having changed the scale, dpi, zoom factor, spacing or margins.

This function returns True if the total geometry has changed.

updatePageSizes()

Compute the correct size of every Page.

computeGeometry()

Return the total geometry (position and size) of the layout.

In most cases the implementation of this method is sufficient: it computes the bounding rectangle of all Pages and adds the margin.

pos2offset(*pos*)

Return a three-tuple (index, x, y).

The index refers to a page in the layout, or nowhere if -1. The x and y refer to a spot on the page (or layout if empty) in the range 0..1. You can use it to store a certain position and restore it after changing the zoom e.g.

offset2pos(*offset*)

Return the pos on the layout for the specified offset.

The offset is a three-tuple like returned by pos2offset().

displayPages()

Return the pages that are to be displayed.

currentPageSetSlice()

Return a slice object describing the current page set.

pageSets()

Return a list of (count, length) tuples.

Every count is the number of page sets of that length. The list is created by the LayoutEngine.pageSets() method.

pageSetCount()

Return the number of page sets.

pageSet(*index*)

Return the page set containing page at index.

engine = <qpageview.layout.LayoutEngine object>

class LayoutEngine

Bases: object

A LayoutEngine takes care of the actual layout process.

A PageLayout has its LayoutEngine in the *engine* attribute. Putting this functionality in a separate object makes it easier to alter the behaviour of a layout without changing all the user-set options and added Pages.

The default implementation of LayoutEngine puts pages in a horizontal or vertical row.

You can override grid() to implement a different behaviour, and you can override pageSets() to get a different behaviour in non-continuous mode.

If there are multiple rows or columns, every row is as high as the highest page it contains, and every column is as wide as its widest page. You can set the attributes evenWidths and/or evenHeights to True if you want all columns to have the same width, and/or respectively, the rows the same height.

zoomToFit = True

orientation = None

evenWidths = False

evenHeights = False

grid(*layout*)

Return a three-tuple (ncols, nrows, prepend).

ncols is the number of columns the layout will contain, nrows the number of rows; and prepend if the number of empty positions that the layout wants, when the first row has less pages.

pages(*layout*, *ncols*, *nrows*, *prepend=0*)

Yield the layout's pages in a grid: (page, (x, y)).

If prepend > 0, that number of first grid positions will remain unused. This can be used for layouts that have less pages in the first row.

dimensions(*layout*, *ncols*, *nrows*, *prepend=0*)

Return two lists: columnwidths and rowheights.

The width and height are page dimensions, without page margin.

updatePagePositions(*layout*)

Performs the positioning of the pages. Don't call on empty layout.

fit(*layout*, *size*, *mode*)

Called by PageLayout.fit().

zoomFitWidth(*layout*, *width*)

Return the zoom factor this layout would need to fit in the width.

This method is called by fit(). The default implementation returns a suitable zoom factor for the widest Page.

zoomFitHeight(*layout*, *height*)

Return the zoom factor this layout would need to fit in the height.

This method is called by fit(). The default implementation returns a suitable zoom factor for the highest Page.

pageSets(*count*)

Return a list of (count, length) tuples.

Every count is the number of page sets of that length. When the layout is in non-continuous mode, it displays only a single page set at a time. For most layout engines, a page set is just one Page, but for column- based layouts other values make sense.

class RowLayoutEngine

Bases: *qpageview.layout.LayoutEngine*

A layout engine that orders pages in rows.

Additional instance attributes:

pagesPerRow = 2, the number of pages to display in a row *pagesFirstRow* = 1, the number of pages to display in the first row *fitAllColumns* = True, whether "fit width" uses all columns

In non-continuous mode, this layout engine displayes a row of pages together. The *orientation* layout attribute is ignored in this layout engine.

pagesPerRow = 2

pagesFirstRow = 1

fitAllColumns = True

orientation = 1

pageSets(*count*)

Return a list of (count, length) tuples respecting our column settings.

grid(*layout*)

Return (ncols, nrows, prepend).

Takes into account the pagesPerRow and pagesFirstRow instance variables. If desired, prepends empty positions so the first row contains less pages than the column width.

zoomFitWidth(*layout*, *width*)

Reimplemented to respect the fitAllColumns setting.

class RasterLayoutEngine

Bases: *qpageview.layout.LayoutEngine*

A layout engine that aligns the pages in a grid.

This layout does not zoom to fit, but changes the number of columns and rows according to the available space. FitBoth is handled like FitWidth.

zoomToFit = False

fit(*layout*, *size*, *mode*)

Reimplemented.

grid(*layout*)

Return a grid that would fit in the layout.

2.5.13 The link module

Generic Link class and handling of links (clickable areas on a Page).

The link area is in coordinates between 0.0 and 1.0, like Poppler does it. This way we can easily compute where the link area is on a page in different sizes or rotations.

class Area(*left*, *top*, *right*, *bottom*)

Bases: `tuple`

bottom

Alias for field number 3

left

Alias for field number 0

right

Alias for field number 2

top

Alias for field number 1

class Link(*left*, *top*, *right*, *bottom*, *url=None*, *tooltip=None*)

Bases: `object`

area = Area(left=0, top=0, right=0, bottom=0)

url = ''

tooltip = ''

rect()

Return the area attribute as a QRectF().

class Links(*objects=None*)

Bases: *qpageview.rectangles.Rectangles*

Manages a list of Link objects.

See the rectangles documentation for how to access the links.

get_coords(*link*)

You should implement this method.

The result should be a four-tuple with the coordinates of the rectangle the object represents (x, y, x2, y2). These are requested only once. x should be < x2 and y should be < y2.

class LinkViewMixin(*parent=None*, ***kwds*)

Bases: `object`

Mixin class to enhance view.View with link capabilities.

linkHovered

(page, link) emitted when the user hovers a link

linkLeft

(no args) emitted when the user does not hover a link anymore

linkClicked

(event, page, link) emitted when the user clicks a link

linkHelpRequested

(event, page, link) emitted when a What's This or Toolip is requested. The event's type determines the type of this help event.

linksEnabled = True

whether to actually enable Link handling

setLinkHighlighter(*highlighter*)

Sets a Highlighter (see highlight.py) to highlight a link on hover.

Use None to remove an active Highlighter. By default no highlighter is set to highlight links on hover.

To be able to actually *use* highlighting, be sure to also mix in the HighlightViewMixin class from the highlight module.

linkHighlighter()

Return the currently set Highlighter, if any.

By default no highlighter is set to highlight links on hover, and None is returned in that case.

adjustCursor(*pos*)

Adjust the cursor if pos is on a link (and linksEnabled is True).

Also emits signals when the cursor enters or leaves a link.

linkAt(*pos*)

If the pos (in the viewport) is over a link, return a (page, link) tuple.

Otherwise returns (None, None).

linkHoverEnter(*page*, *link*)

Called when the mouse hovers over a link.

The default implementation emits the linkHovered(page, link) signal, sets a pointing hand mouse cursor, and, if a Highlighter was set using setLinkHighlighter(), highlights the link. You can reimplement this method to do something different.

linkHoverLeave()

Called when the mouse does not hover a link anymore.

The default implementation emits the linkLeft() signal, sets a default mouse cursor, and, if a Highlighter was set using setLinkHighlighter(), removes the highlighting of the current link. You can reimplement this method to do something different.

linkClickEvent(*ev*, *page*, *link*)

Called when a link is clicked.

The default implementation emits the linkClicked(event, page, link) signal. The event can be used for things like determining which button was used, and which keyboard modifiers were in effect.

linkHelpEvent(*ev*, *page*, *link*)

Called when a ToolTip or WhatsThis wants to appear.

The default implementation emits the linkHelpRequested(event, page, link) signal. Using the event you can find the position, and the type of the help event.

event(*ev*)

Reimplemented to handle HelpEvent for links.

mousePressEvent(*ev*)

Implemented to detect clicking a link and calling linkClickEvent().

leaveEvent(*ev*)

Implemented to leave a link, might there still be one hovered.

2.5.14 The locking module

Manages locking access (across threads) to any object.

Use it for example to lock access to Poppler.Document instances.

lock(*item*)

Return a threading.RLock instance for the given item.

Use:

with lock(document): do_something

2.5.15 The magnifier module

The Magnifier magnifies a part of the displayed document.

class Magnifier

Bases: `PyQt5.QtWidgets.QWidget`

A Magnifier is added to a View with view.setMagnifier().

It is shown when a mouse button is pressed together with a modifier (by default Ctrl). It can then be resized by moving the mouse is with two buttons pressed, or by wheeling with resizemodifier pressed.

Its size can be changed with resize() and the scale (defaulting to 3.0) with setScale().

If can also be shown programatically with the show() method. In this case it can be dragged with the left mouse button.

Wheel zooming with the modifier (by default Ctrl) zooms the magnifier.

Instance attributes:

showmodifier: the modifier to popup (Qt.ControlModifier)

zoommodifier: the modifier to wheel zoom (Qt.ControlModifier)

resizemodifier: the key to press for wheel resizing (Qt.ShiftModifier)

showbutton: the mouse button causing the magnifier to popup (by default Qt.LeftButton)

resizebutton: the extra mouse button to be pressed when resizing the magnifier (by default Qt.RightButton)

MAX_EXTRA_ZOOM: the maximum zoom (relative to the View's maximum zoom level)

showmodifier = 67108864

zoommodifier = 67108864

resizemodifier = 33554432

showbutton = 1

resizebutton = 2

MAX_EXTRA_ZOOM = 1.25

MIN_SIZE = 50

MAX_SIZE = 640

moveCenter(*pos*)
Called by the View, centers the widget on the given QPoint.

setScale(*scale*)
Sets the scale, relative to the dislayed size in the View.

scale()
Returns the scale, defaulting to 3.0 (=300%).

startShortDrag(*pos*)
Start a short drag (e.g. on ctrl+click).

endShortDrag()
End a short drag.

startLongDrag(*pos*)
Start a long drag (when we are already visible and then dragged).

endLongDrag()
End a long drag.

resizeEvent(*ev*)
Called on resize, sets our circular mask.

moveEvent(*ev*)
Called on move, updates the contents.

eventFilter(*viewport*, *ev*)
Handle events on the viewport of the View.

mousePressEvent(*ev*)
Start dragging the magnifier.

mouseMoveEvent(*ev*)
Move the magnifier if we were dragging it.

mouseReleaseEvent(*ev*)
The button is released, stop moving ourselves.

wheelEvent(*ev*)
Implement zooming the magnifying glass.

paintEvent(*ev*)
Called when paint is needed, finds out which page to magnify.

drawBorder(*painter*)
Draw a nice looking glass border.

repaintPage(*page*)
Called when a Page was rendered in the background.

2.5.16 The multipage module

A MultiPage has no contents itself (but it has a size!), and renders a list of embedded pages.

The MultiPageRenderer has the same interface as an ordinary renderer, but defers rendering to the renderer of the embedded pages.

class MultiPage(*renderer=None*)

Bases: *qpageview.page.AbstractRenderedPage*

A special Page that has a list of embedded sub pages.

The sub pages are in the pages attribute, the first one is on top.

The position and size of the embedded pages is set in the updateSize() method, which is inherited from AbstractPage. By default all sub pages are centered in their natural size.

Rotation of sub pages is relative to the MultiPage.

The *scalePages* instance attribute can be used to multiply the zoomfactor for the sub pages.

The *opaquePages* instance attribute optimizes some procedures when set to True (i.e. it prevents rendering sub pages that are hidden below others).

By default, only links in the first sub page are handled. Set *linksOnlyFirstSubPage* to False if you want links in all sub pages.

scalePages = 1.0

opaquePages = True

linksOnlyFirstSubPage = True

renderer = <qpageview.multipage.MultiPageRenderer object>

classmethod createPages(*pageLists*, *renderer=None*, *pad=<class 'qpageview.page.BlankPage'>*)

Yield pages, taking each page from every pageList.

If pad is given and is not None, it is a callable that instantiates blank pages, to pad the shorter pageLists with. In that case, the returned list of pages has the same length as the longest pageList given. If pad is None, the returned list of pages has the same length as the shortest pageList given.

copy(*owner=None*, *matrix=None*)

Reimplemented to also copy the sub pages.

updateSize(*dpiX*, *dpiY*, *zoomFactor*)

Reimplemented to also position our sub-pages.

The default implementation of this method zooms the sub pages at the zoom level of the page * self.scalePages.

updatePagePositions()

Called by updateSize(), set the page positions.

The default implementation of this method centers the pages.

visiblePagesAt(*rect*)

Yield (page, rect) for all subpages.

The rect may be invalid when opaquePages is False. If opaquePages is True, pages outside rect or hidden below others are exclued. The yielded rect is always valid in that case.

printablePagesAt(*rect*)

Yield (page, matrix) for all subpages that are visible in rect.

If opaquePages is True, excludes pages outside rect or hidden below others. The matrix (QTransform) describes the transformation from the page to the sub page. Rect is in original coordinates, as with the print() method.

print(*painter*, *rect=None*, *paperColor=None*)

Prints our sub pages.

text(*rect*)

Reimplemented to get text from sub pages.

linksAt(*point*)

Reimplemented to find links in sub pages.

linksIn(*rect*)

Reimplemented to find links in sub pages.

linkRect(*link*)

Reimplemented to get correct area on the page the link belongs to.

class MultiPageDocument(*sources=()*, *renderer=None*)

Bases: *qpageview.document.MultiSourceDocument*

A Document that combines pages from different documents.

pageClass

alias of *qpageview.multipage.MultiPage*

createPages()

Implement this method to create and yield the pages.

This method is only called once. After altering filename,-s or source,-s, or invalidate(), it is called again.

class MultiPageRenderer(*cache=None*)

Bases: *qpageview.render.AbstractRenderer*

A renderer that interfaces with the renderers of the sub pages of a MultiPage.

update(*page*, *device*, *rect*, *callback=None*)

Reimplemented to check/rerender (if needed) all sub pages.

paint(*page*, *painter*, *rect*, *callback=None*)

Reimplemented to paint all the sub pages on top of each other.

image(*page*, *rect*, *dpiX*, *dpiY*, *paperColor*)

Return a QImage of the specified rectangle, of all images combined.

unschedule(*pages*, *callback*)

Reimplemented to unschedule all sub pages.

invalidate(*pages*)

Reimplemented to invalidate the base and overlay pages.

combine(*painter*, *images*)

Paints images on the painter.

Each image is a tuple(QPoint, QPixmap), describing where to draw. The image on top is first, so drawing should start with the last.

class CallBack(*origcallable*, *page*)

Bases: object

A wrapper for a callable that is called with the original Page.

2.5.17 The page module

A Page is responsible for drawing a page inside a PageLayout.

class AbstractPage

Bases: *qpageview.util.Rectangular*

A Page is a rectangle that is positioned in a PageLayout.

A Page represents one page, added to a PageLayout that is displayed in a View. Although there is no mechanism to enforce it, a Page is normally only used in one PageLayout at a time.

A Page has instance attributes:

- that normally do not change during its lifetime:

  pageWidth the original width (by default in points, *dpi* is 72.0

  pageHeight the original height but can be changed at class level)

- that can be modified by the user (having defaults at the class level):

  scaleX the scale in X-direction of the original page (1.0)

  scaleY the scale in Y-direction of the original page (1.0)

  rotation the rotation (Rotate_0)

  z the z-index (0) (only relevant when pages overlap)

  paperColor the paper color (None). If None, the renderer's paperColor is used.

- and that are set by the layout when computing the size and positioning the pages:

  x the position x-coordinate

  y the position y-coordinate

  width the width in pixels

  height the height in pixels

  computedRotation the rotation in which finally to render

The class variable *dpi* is 72.0 by default but can be set to a different value depending on the page type. E.g. for Svg pages 90 or 96 makes sense.

renderer = None

dpi = 72.0

pageWidth = 595.28

pageHeight = 841.89

z = 0

rotation = 0

computedRotation = 0

scaleX = 1.0

scaleY = 1.0

paperColor = None

classmethod load(*filename*, *renderer=None*)

Implement this to yield one or more pages by reading the file.

The renderer may be None, and not all page types use a renderer. The filename may be a string or a QByteArray object containing the data.

classmethod loadFiles(*filenames*, *renderer=None*)

Load multiple files, yielding Page instances of this type.

copy(*owner=None*, *matrix=None*)

Return a copy of the page with the same instance attributes.

If owner is specified, the copy is weakly cached for that owner and returned next time. All instance attribute will be updated each time. If matrix is specified, it should be a QTransform, and it will be used to map the geometry of the original to the (cached) copy before it is returned.

setPageSize(*sizef*)

Set our natural page size (QSizeF).

Normally this is done in the constructor, based on the page we need to render.

By default the page size is assumed to be in points, 1/72 of an inch. You can set the *dpi* class variable to use a different unit.

pageSize()

Return our natural page size (QSizeF).

By default the page size is assumed to be in points, 1/72 of an inch. You can set the *dpi* class variable to use a different unit.

pageRect()

Return QRectF(0, 0, pageWidth, pageHeight).

transform(*width=None*, *height=None*)

Return a QTransform, converting an original area to page coordinates.

The *width* and *height* refer to the original (unrotated) width and height of the page's contents, and default to pageWidth and pageHeight.

defaultSize()

Return the pageSize() scaled and rotated (if needed).

Based on scaleX, scaleY, and computedRotation attributes.

updateSize(*dpiX*, *dpiY*, *zoomFactor*)

Set the width and height attributes of the page.

This size is computed based on the page's natural size, dpi, scale and computedRotation attribute; and the supplied dpiX, dpiY, and zoomFactor.

zoomForWidth(*width*, *rotation*, *dpiX*)

Return the zoom we need to display ourselves at the given width.

zoomForHeight(*height*, *rotation*, *dpiY*)

Return the zoom we need to display ourselves at the given height.

paint(*painter*, *rect*, *callback=None*)

Implement this to paint our Page.

The View calls this method in the paint event. If you can't paint quickly, just return and schedule an image to be rendered in the background. If a callback is specified, it is called when the image is ready with the page as argument.

print(*painter*, *rect=None*, *paperColor=None*)

Implement this to paint a page for printing.

The difference with paint() and image() is that the rect (QRectF) supplied to print() is not in the Page coordinates, but in the original pageSize() and unrotated. The painter has been prepared for scale and rotation.

If rect is None, the full pageRect() is used.

output(*device*, *rect=None*, *paperColor=None*)

Paint specified rectangle (or the whole page) to the paint device.

The page is rotated and scaled, and the resolution of the paint device is used in case pixelbased images need to be generated. But where possible, vector painting is used.

This method uses *print()* to do the actual painting to the paint device. If paperColor is not given, no background is printed normally.

image(*rect=None*, *dpiX=None*, *dpiY=None*, *paperColor=None*)

Implement this to return a QImage of the specified rectangle.

The rectangle is relative to our top-left position. dpiX defaults to our default dpi and dpiY defaults to dpiX.

pdf(*filename*, *rect=None*, *resolution=72.0*, *paperColor=None*)

Create a PDF file for the selected rect or the whole page.

The filename may be a string or a QIODevice object. The rectangle is relative to our top-left position. Normally vector graphics are rendered, but in cases where that is not possible, the resolution will be used to determine the DPI for the generated rendering.

eps(*filename*, *rect=None*, *resolution=72.0*, *paperColor=None*)

Create a EPS (Encapsulated Postscript) file for the selected rect or the whole page.

This needs the popplerqt5 module. The filename may be a string or a QIODevice object. The rectangle is relative to our top-left position. Normally vector graphics are rendered, but in cases where that is not possible, the resolution will be used to determine the DPI for the generated rendering.

svg(*filename*, *rect=None*, *resolution=72.0*, *paperColor=None*)

Create a SVG file for the selected rect or the whole page.

The filename may be a string or a QIODevice object. The rectangle is relative to our top-left position. Normally vector graphics are rendered, but in cases where that is not possible, the resolution will be used to determine the DPI for the generated rendering.

pixmap(*rect=None*, *size=100*, *paperColor=None*)

Return a QPixmap, scaled so that width or height doesn't exceed size.

Uses the *image()* method to get the image, and converts that to a QPixmap.

mutex()

Return an object that should be locked when rendering the page.

Page are guaranteed not to be rendered at the same time when they return the same mutex object. By default, None is returned.

group()

Return the group the page belongs to.

This could be some document structure, so that different Page objects could refer to the same graphical contents, preventing double caching.

This object is used together with the value returned by ident() as a key to cache the page. The idea is that the contents of the page are uniquely identified by the objects returned by group() and ident().

This way, when the same document is opened in multiple page instances, only one copy resides in the (global) cache.

By default, the page object itself is returned.

ident()

Return a value that identifies the page within the group returned by group().

By default, None is returned.

mapToPage(*width=None*, *height=None*)

Return a MapToPage object, that can map original to Page coordinates.

The *width* and *height* refer to the original (unrotated) width and height of the page's contents, and default to pageWidth and pageHeight.

mapFromPage(*width=None*, *height=None*)

Return a MapFromPage object, that can map Page to original coordinates.

The *width* and *height* refer to the original (unrotated) width and height of the page's contents, and default to pageWidth and pageHeight.

text(*rect*)

Implement this method to get the text at the specified rectangle.

The rectangle should be in page coordinates. The default implementation simply returns an empty string.

getLinks()

Implement this method to load our links.

links()

Return the Links object, containing Link objects.

Every Link denotes a clickable area on a Page, in coordinates 0.0-1.0. The Links object makes it possible to quickly find a link on a Page. This is cached after the first request, you should implement the getLinks() method to load the links.

linksAt(*point*)

Return a list of zero or more links touched by QPoint point.

The point is in page coordinates. The list is sorted with the smallest rectangle first.

linksIn(*rect*)

Return an unordered set of links enclosed in rectangle.

The rectangle is in page coordinates.

linkRect(*link*)

Return a QRect encompassing the linkArea of a link in coordinates of our page.

class AbstractRenderedPage(*renderer=None*)

Bases: *qpageview.page.AbstractPage*

A Page that has a renderer that performs caching and painting.

The renderer lives in the renderer attribute.

paint(*painter*, *rect*, *callback=None*)

Reimplement this to paint our Page.

The View calls this method in the paint event. If you can't paint quickly, just return and schedule an image to be rendered in the background. If a callback is specified, it is called when the image is ready with the page as argument.

By default, this method calls the renderer's *paint()* method.

print(*painter*, *rect=None*, *paperColor=None*)

Paint a page for printing.

The difference with *paint()* and *image()* is that the rect (QRectF) supplied to print() is not in the Page coordinates, but in the original pageSize() and unrotated. The painter has been prepared for scale and rotation.

If rect is None, the full pageRect() is used. This method calls the renderer's draw() method.

image(*rect=None*, *dpiX=None*, *dpiY=None*, *paperColor=None*)

Returns a QImage of the specified rectangle.

The rectangle is relative to our top-left position. dpiX defaults to our default dpi and dpiY defaults to dpiX. This implementation calls the renderer to generate the image. The image is not cached.

class BlankPage

Bases: *qpageview.page.AbstractPage*

A blank page.

paint(*painter*, *rect*, *callback=None*)

Paint blank page in the View.

print(*painter*, *rect=None*, *paperColor=None*)

Paint blank page for printing.

image(*rect=None*, *dpiX=None*, *dpiY=None*, *paperColor=None*)

Return a blank image.

class ImagePrintPageMixin

Bases: `object`

A Page mixin that implements print() using the image() method.

This can be used e.g. for compositing pages, which does not work well when painting to a PDF, a printer or a SVG generator.

print(*painter*, *rect=None*, *paperColor=None*)

Print using the image() method.

2.5.18 The pkginfo module

Meta-information about the qpageview package.

This information is used by the install script.

class Version(*major*, *minor*, *patch*)

Bases: `tuple`

major

Alias for field number 0

minor

Alias for field number 1

patch

Alias for field number 2

name = 'qpageview'

name of the package

version = Version(major=0, minor=6, patch=2)

the current version

version_string = '0.6.2'

the current version as a string

description = 'Widget to display page-based documents for Qt5/PyQt5'

short description

long_description = 'The qpageview package provides a Python library to display page-based documents, such as PDF and possibly other formats.'

long description

maintainer = 'Wilbert Berendsen'

maintainer name

maintainer_email = 'info@frescobaldi.org'

maintainer email

url = 'https://github.com/frescobaldi/qpageview'

homepage

license = 'GPL v3'

license

copyright_year = '2020-2022'

copyright year

2.5.19 The poppler module

Interface with popplerqt5, popplerqt5-specific classes etc.

This module depends on popplerqt5, although it can be imported when popplerqt5 is not available.

You need this module to display PDF documents.

class Link(*linkobj*)

Bases: *qpageview.link.Link*

A Link that encapsulates a Poppler.Link object.

property url

The url the link points to.

class PopplerPage(*document*, *pageNumber*, *renderer=None*)

Bases: *qpageview.page.AbstractRenderedPage*

A Page capable of displaying one page of a Poppler.Document instance.

It has two additional instance attributes:

document: the Poppler.Document instance *pageNumber*: the page number to render

classmethod loadPopplerDocument(*document*, *renderer=None*, *pageSlice=None*)

Convenience class method yielding instances of this class.

The Page instances are created from the document, in page number order. The specified Renderer is used, or else the global poppler renderer. If pageSlice is given, it should be a slice object and only those pages are then loaded.

classmethod load(*filename*, *renderer=None*)

Load a Poppler document, and yield of instances of this class.

The filename can also be a QByteArray or a popplerqt5.Poppler.Document instance. The specified Renderer is used, or else the global poppler renderer.

mutex()

No two pages of same Poppler document are rendered at the same time.

group()

Reimplemented to return the Poppler document our page displays a page from.

ident()

Reimplemented to return the page number of this page.

text(*rect*)

Returns text inside rectangle.

links()

Reimplemented to use a different caching mechanism.

renderer = <qpageview.poppler.PopplerRenderer object>

class PopplerDocument(*source=None*, *renderer=None*)

Bases: *qpageview.document.SingleSourceDocument*

A lazily loaded Poppler (PDF) document.

pageClass

alias of *qpageview.poppler.PopplerPage*

invalidate()

Reimplemented to clear the Poppler Document reference.

createPages()

Implement this method to create and yield the pages.

This method is only called once. After altering filename,-s or source,-s, or invalidate(), it is called again.

document()

Return the Poppler Document object.

Returns None if no source was yet set, and False if loading failed.

class PopplerRenderer(*cache=None*)

Bases: *qpageview.render.AbstractRenderer*

renderBackend = 0

printRenderBackend = 0

oversampleThreshold = 96

render(*page*, *key*, *tile*, *paperColor=None*)

Generate an image for the Page referred to by key.

setRenderHints(*doc*)

Set the poppler render hints we want to set.

setup(*doc*, *backend=None*, *paperColor=None*)

Use the poppler document in context, properly configured and locked.

render_poppler_image(*doc*, *pageNum*, *xres=72.0*, *yres=72.0*, *x=- 1*, *y=- 1*, *w=- 1*, *h=- 1*, *rotate=0*, *paperColor=None*)

Render an image, almost like calling page.renderToImage().

The document is properly locked during rendering and render options are set.

draw(*page*, *painter*, *key*, *tile*, *paperColor=None*)

Draw a tile on the painter.

The painter is already at the right position and rotation. For the Poppler page and renderer, draw() is only used for printing. (See AbstractPage.print().)

load(*source*)

Load a Poppler document.

Source may be:

- a Poppler document, which is then simply returned :-)
- a filename
- q QByteArray instance.

Returns None if popplerqt5 is not available or the document could not be loaded.

2.5.20 The printing module

Printing facilities for qpageview.

class PrintJob(*printer*, *pageList*, *parent=None*)

Bases: *qpageview.backgroundjob.Job*

Performs a print job in the background.

Emits the following signals:

progress(pageNumber, num, total) before each Page

finished() when done

progress

aborted = False

setPageList(*pageList*)

Set the pagelist to print.

pageList may be a list of two-tuples (num, page). Otherwise, the pages are numbered from 1 in the progress message. The pages are copied.

work()

Paint the pages to the printer in the background.

class PrintProgressDialog(*job*, *parent=None*)

Bases: PyQt5.QtWidgets.QProgressDialog

A simple progress dialog displaying the printing progress.

showProgress(*page*, *num*, *total*)

Called by the job when printing a page.

jobFinished()

Called when the print job has finished.

showErrorMessage()

Reimplement to show a different or translated error message.

2.5.21 The rectangles module

Manages lists of rectangular objects and quickly finds them.

class Rectangles(*objects=None*)

Bases: object

Manages a list of rectangular objects and quickly finds objects at some point, in some rectangle or intersecting some rectangle.

The implementation uses four lists of the objects sorted on either coordinate, so retrieval is fast.

Bulk adding is done in the constructor or via the bulk_add() method (which clears the indexes, that are recreated on first search). Single objects can be added and deleted, keeping the indexes, but that's slower.

You should inherit from this class and implement the method get_coords(obj) to get the rectangle of the object (x, y, x2, y2). These are requested only once. x should be < x2 and y should be < y2.

get_coords(*obj*)

You should implement this method.

The result should be a four-tuple with the coordinates of the rectangle the object represents (x, y, x2, y2). These are requested only once. x should be < x2 and y should be < y2.

add(*obj*)

Adds an object to our list. Keeps the index intact.

bulk_add(*objects*)

Adds many new items to the index using the function given in the constructor.

After this, the index is cleared and recreated on the first search operation.

remove(*obj*)

Removes an object from our list. Keeps the index intact.

clear()

Empties the list of items.

at(*x*, *y*)

Returns a set() of objects that are touched by the given point.

inside(*left*, *top*, *right*, *bottom*)

Returns a set() of objects that are fully in the given rectangle.

intersecting(*left*, *top*, *right*, *bottom*)

Returns a set() of objects intersecting the given rectangle.

width(*obj*)

Return the width of the specified object.

This can be used for sorting a set returned by at(), inside() or intersecting(). For example:

```
for r in sorted(rects.at(10, 20), key=rects.width):
    # ...
```

height(*obj*)

Return the height of the specified object. See also width().

closest(*obj*, *side*)

Returns the object closest to the given one, going to the given side.

nearest(*x*, *y*)

Return the object with the shortest distance to the point x, y.

The point (x, y) is outside the object. Use at() to get objects that touch the point (x, y). If there are no objects, None is returned.

2.5.22 The render module

Infrastructure for rendering and caching Page images.

class Tile(*x*, *y*, *w*, *h*)

Bases: tuple

Describes a tile to render. Most times all coordinates are integers. The needed tiles for a page are yielded by *AbstractRenderer.tiles()*.

h

The height of the tile

w

The width of the tile

x

The x coordinate of the tile

y

The y coordinate of the tile

class Key(*group*, *ident*, *rotation*, *width*, *height*)

Bases: tuple

Identifies a render operation for a Page, returned by *AbstractRenderer.key()*.

group

The *group()* of the page

height

The *height* of the page

ident

The *ident()* of the page

rotation

The *computedRotation* of the page

width

The *width* of the page

class RenderInfo(*images*, *missing*, *key*, *target*, *ratio*)

Bases: tuple

Information about cached or missing rendered tiles to display a rectangular part of a Page at a certain size. Returned by *AbstractRenderer.info()*.

images

the devicepixelratio of the specified paint device

key

the Key returned by *key()*, describing width, height, rotation and identity of the page

missing

a list of Tile instances that are needed but not available in the cache

ratio

Alias for field number 4

target

the rect multiplied by the ratio

class AbstractRenderer(*cache=None*)

Bases: object

Handle rendering and caching of images.

A renderer can be assigned to the renderer attribute of a Page and takes care for generating, caching and updating the images needed for display of the Page at different sizes.

You can use a renderer for as many Page instances as you like. You can use one global renderer in your application or more, depending on how you use the qpageview package.

You must inherit from this class and at least implement the render() or the draw() method.

Instance attributes:

paperColor Paper color. If possible this background color is used when rendering the pages, also for temporary drawings when a page has to be rendered. If a Page specifies its own paperColor, that color prevails.

imageFormat QImage format to use (if possible). Default is QImage.Format_ARGB32_Premultiplied

antialiasing True by default. Whether to antialias graphics. (Most Renderers antialias anyway, even if this is False.)

MAX_TILE_WIDTH = 2400

MAX_TILE_HEIGHT = 1600

paperColor = <PyQt5.QtGui.QColor object>

imageFormat = 6

antialiasing = True

cache = <qpageview.cache.ImageCache object>

copy()

Return a copy of the renderer, with always a new cache.

static key(*page*, *ratio*)

Return a five-tuple Key describing the page.

The ratio is a device pixel ratio; width and height are multiplied with this value, to render and cache an image correctly on high- density displays.

This is used for rendering and caching. It is never stored as is. The cache can store the group object using a weak reference. The tuple contains the following values:

group the object returned by `page.group()`

ident the value returned by `page.ident()`

rotation `page.computedRotation`

width `page.width * ratio`

height `page.height * ratio`

tiles(*width*, *height*)

Yield four-tuples Tile(x, y, w, h) describing the tiles to render.

map(*key*, *box*)

Return a QTransform converting from Key coordinates to a box.

The box should be a QRectF or QRect, and describes the original area of the page. The returned matrix can be used to convert e.g. tile coordinates to the position on the original page.

image(*page*, *rect*, *dpiX*, *dpiY*, *paperColor*)

Returns a QImage of the specified rectangle on the Page.

The rectangle is relative to the top-left position. The image is not cached.

render(*page*, *key*, *tile*, *paperColor=None*)

Generate a QImage for tile of the Page.

The width, height and rotation to render at should be taken from the key, as the page could be resized or rotated in the mean time.

The default implementation prepares the image, a painter and then calls draw() to actually draw the contents.

If the paperColor is not specified, it will be read from the Page's paperColor attribute (if not None) or else from the renderer's paperColor attribute.

draw(*page*, *painter*, *key*, *tile*, *paperColor=None*)

Draw the page contents; implement at least this method.

The painter is already at the top-left position and the correct rotation. You should convert the tile to the original area on the page, you can use the map() method for that. You can draw in tile/key coordinates. Don't use width, height and rotation from the Page object, as it could have been resized or rotated in the mean time.

The paperColor can be speficied, but it is not needed to paint it: by default the render() method already fills the image, and when drawing on a printer, painting the background is normally not desired.

info(*page*, *device*, *rect*)

Return a namedtuple RenderInfo(images, missing, key, target, ratio).

images is a list of tuples (tile, image) that are available in the cache; missing is a list of Tile instances that are not available in the cache; key is the Key returned by key(), describing width, height, rotation and identity of the page; target is the rect multiplied by the ratio; which is the devicepixelratio of the specified paint device.

update(*page*, *device*, *rect*, *callback=None*)

Check if a page can be painted on the device without waiting.

Return True if that is the case. Otherwise schedules missing tiles for rendering and calls the callback each time one tile if finished.

paint(*page*, *painter*, *rect*, *callback=None*)

Paint a page, using images from the cache.

page: the Page to draw

painter: the QPainter to use to draw

rect: the region to draw, relative to the topleft of the page.

callback: if specified, a callable accepting the *page* argument. Typically this should be used to trigger a repaint of the view.

The Page calls this method by default in its *paint()* method. This method tries to fetch an image from the cache and paint that. If no image is available, render() is called in the background to generate one. If it is ready, the callback is called with the Page as argument. An interim image may be painted in the meantime (e.g. scaled from another size).

schedule(*page*, *key*, *tiles*, *callback*)

Schedule a new rendering job for the specified tiles of the page.

If this page has already a job pending, the callback is added to the pending job.

job(*page*, *key*, *tile*)

Return a new *Job* tailored for this tile.

unschedule(*pages*, *callback*)

Unschedule a possible pending rendering job for the given pages.

If the pending job has no other callbacks left, it is removed, unless it is running.

invalidate(*pages*)

Delete the cached images for the given pages.

checkstart()

Check whether there are jobs that need to be started.

This method is called by the schedule() method, and by the finish() method when a job finishes, so that the number of running jobs never exceeds *maxjobs*.

exception(*exctype*, *excvalue*, *exctb*)

Called when an exception has occurred in a background rendering job.

The default implementation prints a traceback to stderr.

2.5.23 The rubberband module

Rubberband selection in a View.

class Rubberband

Bases: `PyQt5.QtWidgets.QWidget`

A Rubberband to select a rectangular region.

A Rubberband is added to a View with view.setRubberband().

The Rubberband lets the user select a rectangular region. When the selection is changed, the *selectionChanged* signal is emitted, having the selection rectangle in layout coordinates as argument.

Instance variables:

showbutton (**Qt.RightButton**) the button used to drag a new rectangle

dragbutton (**Qt.LeftButton**) the button to alter an existing rectangle

trackSelection (**False**) whether to continuously emit selectionChanged(). When True, `selectionChanged()` is emitted on every change, when False, the signal is only emitted when the mouse button is released.

selectionChanged

showbutton = 2

dragbutton = 1

trackSelection = False

paintEvent(*self*, *QPaintEvent*)

edge(*point*)

Return the edge where the point touches our geometry.

adjustCursor(*edge*)

Sets the cursor shape when we are at edge.

hasSelection()

Return True when there is a selection.

selection()

Return our selection rectangle, relative to the view's layout position.

selectedPages()

Yield tuples (page, rect) describing the selection.

Every rect is intersected with the page rect and translated to the page's position.

selectedPage()

Returns (page, rect) if there is a selection.

If the selection contains more pages, the largest intersection is chosen. If no meaningful area is selected, (None, None) is returned.

selectedImage(*resolution=None*, *paperColor=None*)

Returns an image of the selected part on a Page.

If resolution is None, the displayed size is chosen. Otherwise, the resolution is an integer, interpreted as DPI (dots per inch).

selectedText()

Return the text found in the selection, as far as the pages support it.

selectedLinks()

Yield tuples (page, links) for every page in the selection.

links is a non-empty set() of Link instances on that page that intersect with the selection.

setSelection(*rect*)

Sets the selection, the rectangle should be relative to the view's layout position.

clearSelection()

Hide ourselves and clear the selection.

scrollBy(*diff*)

Called by the View when scrolling.

startDrag(*pos*, *button*)

Start dragging the rubberband.

drag(*pos*)

Continue dragging the rubberband, scrolling the View if necessary.

dragBy(*diff*)

Drag by diff (QPoint).

stopDrag()

Stop dragging the rubberband.

slotZoomChanged(*zoom*)

Called when the zooming in the view changes, resizes ourselves.

eventFilter(*viewport*, *ev*)

Act on events in the viewport:

- keep on the same place when the viewport resizes
- start dragging the selection if showbutton clicked (preventing the contextmenu if the showbutton is the right button)
- end a drag on mousebutton release, if that button would have shown the context menu, show it on button release.

mousePressEvent(*ev*)

Can start a new drag when we are clicked ourselves.

mouseMoveEvent(*ev*)

Move if we are dragging; show the correct cursor shape on the edges.

mouseReleaseEvent(*ev*)

End a self-initiated drag; if the right button was used; send a context menu event.

2.5.24 The scrollarea module

ScrollArea, that supports kinetic scrolling and other features.

class ScrollArea(*parent=None*, ***kwds*)

Bases: PyQt5.QtWidgets.QAbstractScrollArea

A scroll area that supports kinetic scrolling and other features.

alignment = 132

how to align the scrolled area if smaller than the viewport (Qt.AlignCenter)

scrollupdatespersec = 50

how many scroll updates to draw per second (50, 50 is recommended).

kineticScrollingEnabled = True

whether the mouse wheel and PgUp/PgDn keys etc use kinetic scrolling (True)

draggingEnabled = True

If enabled, the user can drag the contents of the scrollarea to move it with the mouse.

setAreaSize(*size*)

Updates the scrollbars to be able to display an area of this size.

areaSize()

Return the size of the area as set by setAreaSize().

areaPos()

Return the position of the area relative to the viewport.

The alignment attribute is taken into account when the area is smaller than the viewport (horizontally and/or vertically).

visibleArea()

Return a rectangle describing the part of the area that is visible.

offsetToEnsureVisible(*rect*)

Return an offset QPoint with the minimal scroll to make rect visible.

If the rect is too large, it is positioned top-left.

ensureVisible(*rect*, *margins=None*, *allowKinetic=True*)

Performs the minimal scroll to make rect visible.

If the rect is not completely visible it is scrolled into view, adding the margins if given (a QMargins instance). If allowKinetic is False, immediately jumps to the position, otherwise scrolls smoothly (if kinetic scrolling is enabled).

scrollOffset()

Return the current scroll offset.

canScrollBy(*diff*)

Does not scroll, but return the actual distance the View would scroll.

diff is a QPoint instance.

scrollForDragging(*pos*)

Slowly scroll the View if pos is close to the edge of the viewport.

Can be used while dragging things.

scrollTo(*pos*)

Scroll the View to get pos (QPoint) in the top left corner (if possible).

Returns the actual distance moved.

scrollBy(*diff*)

Scroll the View diff pixels (QPoint) in x and y direction.

Returns the actual distance moved.

kineticScrollTo(*pos*)

Scroll the View to get pos (QPoint) in the top left corner (if possible).

Returns the actual distance the scroll area will move.

kineticScrollBy(*diff*)

Scroll the View diff pixels (QPoint) in x and y direction.

Returns the actual distance the scroll area will move.

kineticAddDelta(*diff*)

Add diff (QPoint) to an existing kinetic scroll.

If no scroll is active, a new one is started (like kineticScrollBy).

steadyScroll(*diff*)

Start steadily scrolling diff (QPoint) pixels per second.

Stops automatically when the end is reached.

startScrolling(*scroller*)

Begin a scrolling operation using the specified scroller.

stopScrolling()

Stop scrolling.

isScrolling()

Return True if a scrolling movement is active.

remainingScrollTime()

If a kinetic scroll is active, return how many msecs the scroll wil last.

Otherwise, return 0.

isDragging()

Return True if the user is dragging the background.

timerEvent(*ev*)

Implemented to handle the scroll timer.

resizeEvent(*ev*)

Implemented to update the scrollbars to the aera size.

mousePressEvent(*ev*)

Implemented to handle dragging the document with the left button.

mouseMoveEvent(*ev*)

Implemented to handle dragging the document with the left button.

mouseReleaseEvent(*ev*)

Implemented to handle dragging the document with the left button.

wheelEvent(*ev*)

Reimplemented to use kinetic mouse wheel scrolling if enabled.

keyPressEvent(*ev*)

Reimplemented to use kinetic cursor movements.

class Scroller

Bases: `object`

Abstract base class, encapsulates scrolling behaviour.

A Scroller subclass must implement the step() and finished() methods and may define additional methods.

step()

Implement this method to return a QPoint for the current scrolling step.

finished()

Implement this method to return True if scrolling is finished.

class SteadyScroller(*speed*, *updates_per_second*)

Bases: *qpageview.scrollarea.Scroller*

Scrolls the area steadily n pixels per second.

step()

Return a QPoint indicating the diff to scroll in this step.

If this is a QPoint(0, 0) it does not indicate that scrolling has finished. Use finished() for that.

finished()

As this scroller has a constant speed, it never stops.

class KineticScroller

Bases: *qpageview.scrollarea.Scroller*

Scrolls the area with a decreasing speed.

scrollBy(*diff*)

Start a new kinetic scroll of the specified amount.

remainingDistance()

Return the remaining distance.

remainingTicks()

Return the remaining ticks of this scroll.

step()

Return a QPoint indicating the diff to scroll in this step.

finished()

Return True if scrolling is done.

2.5.25 The selector module

SelectorViewMixin class, to mixin with View.

Adds the capability to select or unselect Pages.

class SelectorViewMixin(*parent=None*, ***kwds*)

Bases: `object`

SelectorViewMixin class, to mixin with View.

Adds the capability to select or unselect Pages. Pages are numbered from 1.

Instance variables:

userChangeSelectionModeEnabled = True whether the user can change the selectionMode (by longpressing a page to enable selectionMode, and pressing ESC to leave selectionMode. (Be sure to mix in the *qpageview.util.LongMousePressMixin* class when you want to use the long mouse press event.)

selectionChanged

selectionModeChanged

userChangeSelectionModeEnabled = True

selection()

Return the current list of selected page numbers.

modifySelection()

Context manager that allows changing the selection.

Yields a set, and on exit of the context, stores the modifications and emits the selectionChanged() signal. Used internally by all other methods.

updatePageLayout(*lazy=False*)

Reimplemented to also check the selection.

clearSelection()

Convenience method to clear the selection.

selectAll()

Convenience method to select all pages.

toggleSelection(*pageNumber*)

Toggles the selected state of page number pageNumber.

selectionMode()

Return the current selectionMode (True is enabled, False is disabled).

setSelectionMode(*mode*)

Switch selection mode on or off (True is enabled, False is disabled).

paintEvent(*ev*)

drawSelection(*page*, *painter*)

Draws the state (selected or not) for the page.

mousePressEvent(*ev*)

Reimplemented to check if a checkbox was clicked.

keyPressEvent(*ev*)

Clear the selection and switch off selectionmode with ESC.

longMousePressEvent(*ev*)

Called on long mouse button press, set selectionMode on if enabled.

2.5.26 The shadow module

A View mixin class that draws a nice drop shadow around all pages.

class ShadowViewMixin

Bases: `object`

Mixin class that draws a drop shadow around every Page.

Drawing the drop shadow can be turned off by setting dropShadowEnabled to False.

dropShadowEnabled = True

paintEvent(*ev*)

drawDropShadow(*page*, *painter*, *width*)

Draw a drop shadow of width pixels around the Page.

The painter is already translated to the topleft corner of the Page.

2.5.27 The sidebarview module

SidebarView, a special View with miniatures to use as a sidebar for a View.

Automatically displays all pages in a view in small size, and makes it easier to browse large documents.

class SidebarView(*parent=None*, ***kwds*)

Bases: *qpageview.selector.SelectorViewMixin*, *qpageview.util.LongMousePressMixin*, *qpageview.view.View*

A special View with miniatures to use as a sidebar for a View.

Automatically displays all pages in a view in small size, and makes it easier to browse large documents. Use setView() to connect a View, and it automatically shows the pages, also when the view is changed.

MAX_ZOOM = 1.0

pagingOnScrollEnabled = False

whether to keep track of current page while scrolling

wheelZoomingEnabled = False

whether to enable mouse wheel zooming

firstPageNumber = 1

scrollupdatespersec = 100

how many scroll updates to draw per second (50, 50 is recommended).

autoOrientationEnabled = True

setOrientation(*orientation*)

Reimplemented to also set the corresponding view mode.

setLayoutFontHeight()

Reads the current font height and reserves enough space in the layout.

setView(*view*)

Connects to a View, or disconnects the current view if view is None.

slotLayoutUpdated()

Called when the layout of the connected view is updated.

slotCurrentPageNumberChanged(*num*)

Called when the page number in the connected view changes.

Does not scroll but updates the current page mark in our View.

paintEvent(*ev*)

Reimplemented to print page numbers and a selection box.

wheelEvent(*ev*)

Reimplemented to page instead of scroll.

keyPressEvent(*ev*)

Reimplemented to page instead of scroll.

resizeEvent(*ev*)

Reimplemented to auto-change the orientation if desired.

changeEvent(*ev*)

Reimplemented to set the correct font height for the page numbers.

2.5.28 The svg module

A page that can display a SVG document.

class SvgPage(*svgrenderer*, *renderer=None*)

Bases: *qpageview.page.AbstractRenderedPage*

A page that can display a SVG document.

dpi = 90.0

classmethod load(*filename*, *renderer=None*)

Load a SVG document from filename, which may also be a QByteArray.

Yields only one Page instance, as SVG currently supports one page per file. If the file can't be loaded by the underlying QSvgRenderer, no Page is yielded.

mutex()

Return an object that should be locked when rendering the page.

Page are guaranteed not to be rendered at the same time when they return the same mutex object. By default, None is returned.

group()

Return the group the page belongs to.

This could be some document structure, so that different Page objects could refer to the same graphical contents, preventing double caching.

This object is used together with the value returned by ident() as a key to cache the page. The idea is that the contents of the page are uniquely identified by the objects returned by group() and ident().

This way, when the same document is opened in multiple page instances, only one copy resides in the (global) cache.

By default, the page object itself is returned.

renderer = <qpageview.svg.SvgRenderer object>

class SvgDocument(*sources=()*, *renderer=None*)

Bases: *qpageview.document.MultiSourceDocument*

A Document representing a group of SVG files.

pageClass

alias of *qpageview.svg.SvgPage*

createPages()

Implement this method to create and yield the pages.

This method is only called once. After altering filename,-s or source,-s, or invalidate(), it is called again.

class SvgRenderer(*cache=None*)

Bases: *qpageview.render.AbstractRenderer*

Render SVG pages.

setRenderHints(*painter*)

Sets the renderhints for the painter we want to use.

draw(*page*, *painter*, *key*, *tile*, *paperColor=None*)

Draw the specified tile of the page (coordinates in key) on painter.

2.5.29 The util module

Small utilities and simple base classes for the qpageview module.

class Rectangular

Bases: object

Defines a Qt-inspired and -based interface for rectangular objects.

The attributes x, y, width and height default to 0 at the class level and can be set and read directly.

For convenience, Qt-styled methods are available to access and modify these attributes.

x = 0

y = 0

width = 0

height = 0

setPos(*point*)

Set the x and y coordinates from the given QPoint point.

pos()

Return our x and y coordinates as a QPoint(x, y).

setSize(*size*)

Set the height and width attributes from the given QSize size.

size()

Return the height and width attributes as a QSize(width, height).

setGeometry(*rect*)

Set our x, y, width and height directly from the given QRect.

geometry()

Return our x, y, width and height as a QRect.

rect()

Return QRect(0, 0, width, height).

class MapToPage(*transform*)

Bases: object

Simple class wrapping a QTransform to map rect and point to page coordinates.

rect(*rect*)

Convert QRect or QRectF to a QRect in page coordinates.

point(*point*)

Convert QPointF or QPoint to a QPoint in page coordinates.

class MapFromPage(*transform*)

Bases: *qpageview.util.MapToPage*

Simple class wrapping a QTransform to map rect and point from page to original coordinates.

rect(*rect*)

Convert QRect or QRectF to a QRectF in original coordinates.

point(*point*)

Convert QPointF or QPoint to a QPointF in original coordinates.

class LongMousePressMixin(**args*, ***kwargs*)

Bases: `object`

Mixin class to add support for long mouse press to a QWidget.

To handle a long mouse press event, implement longMousePressEvent().

longMousePressEnabled = True

Whether to enable handling of long mouse presses; set to False to disable

longMousePressTolerance = 3

Allow moving some pixels before a long mouse press is considered a drag

longMousePressTime = 800

How long to presse a mouse button (in msec) for a long press

longMousePressEvent(*ev*)

Implement this to handle a long mouse press event.

timerEvent(*ev*)

Implemented to check for a long mouse button press.

mousePressEvent(*ev*)

Reimplemented to check for a long mouse button press.

mouseMoveEvent(*ev*)

Reimplemented to check for moves during a long press.

mouseReleaseEvent(*ev*)

Reimplemented to cancel a long press.

rotate(*matrix*, *rotation*, *width*, *height*, *dest=False*)

Rotate matrix inside a rectangular area of width x height.

The `matrix` can be a either a QPainter or a QTransform. The `rotation` is 0, 1, 2 or 3, etc. (`Rotate_0`, `Rotate_90`, etc…). If `dest` is True, `width` and `height` refer to the destination, otherwise to the source.

align(*w*, *h*, *ow*, *oh*, *alignment=132*)

Return (x, y) to align a rect w x h in an outer rectangle ow x oh.

The alignment can be a combination of the Qt.Alignment flags. If w > ow, x = -1; and if h > oh, y = -1.

alignrect(*rect*, *point*, *alignment=132*)

Align rect with point according to the alignment.

The alignment can be a combination of the Qt.Alignment flags.

sign(*x*)

Return the sign of x: -1 if x < 0, 0 if x == 0, or 1 if x > 0.

signalsBlocked(**objs*)

Block the pyqtSignals of the given QObjects during the context.

autoCropRect(*image*)

Return a QRect specifying the contents of the QImage.

Edges of the image are trimmed if they have the same color.

tempdir()

Return a temporary directory that is erased on app quit.

2.5.30 The viewactions module

ViewActions provides QActions to control a View.

class ViewActions(**args*, ***kwargs*)

Bases: PyQt5.QtCore.QObject

ViewActions provides QActions to control a View.

Use setView() to connect the actions with a View. If no View is connected, and an action is used; the viewRequested signal is emitted. You can connect this signal and call setView() in the called slot; the action is then performed on the View.

The attribute *smartLayoutOrientationEnabled* (defaulting to True) enables some intuitive behaviour: if set to True, for layout modes that do not make sense in horizontal mode the orientation is automatically set to Vertical; and when the user chooses Horizontal orientation in such modes, the layout mode is set to "single".

smartLayoutOrientationEnabled = True

viewRequested

setView(*view*)

Connects all the actions to the View.

Use None to set no view. If a view was previously set, all connections are removed from that View.

view()

Return the View.

If no View is set, viewRequested is emitted. You can connect to this signal to create a View, and call setView() to use it to perform the requested action.

static names()

Return a tuple of all the names of the actions we support.

createActions()

Creates the actions; called by __init__().

updateFromProperties(*properties*)

Set the actions to the state stored in the given ViewProperties.

connectActions()

Connect our actions with our methods. Called by __init__().

updateActions()

Update the state of the actions not handled in the other update methods.

updatePageLayoutModeActions(*mode*)

Update the state of the layout mode actions.

updateViewModeActions(*mode*)

Update the state of view mode related actions.

updateZoomActions(*factor*)

Update the state of zoom related actions.

updatePagerActions()

Update the state of paging-related actions.

setActionTexts(*_=None*)

Set a default text to all the actions, you may override or translate them.

You may also set tooltip or whatsthis text in this method.

setActionIcons()

Implement this method to set icons to the actions.

setActionShortcuts()

Implement this method to set keyboard shortcuts to the actions.

slotPrint()

slotViewMode(*action*)

slotZoomNatural()

slotZoomOriginal()

slotZoomIn()

slotZoomOut()

slotZoomViewMode(*mode*)

slotZoomFactor(*factor*)

slotRotateLeft()

slotRotateRight()

slotPageLayoutMode(*action*)

slotOrientation(*action*)

slotContinuousMode()

slotReload()

slotPreviousPage()

slotNextPage()

slotSetPageNumber(*num*)

slotMagnifier()

class PagerAction(**args*, ***kwargs*)

Bases: `PyQt5.QtWidgets.QWidgetAction`

PagerAction shows a spinbox widget with the current page number.

When the current page number is changed (by the user or by calling setCurrentPageNumber()) the signal currentPageNumberChanged() is emitted with the new current page number.

You can use the instance or class attributes buttonSymbols, focusPolicy and the displayFormat() method to influence behaviour and appearance of the spinbox widget(s) that is/are created when this action is added to a toolbar.

The displayFormat string should contain the text "{num}". You can also include the string "{total}", so the page count is displayed as well.

currentPageNumberChanged

buttonSymbols = 2

focusPolicy = 2

createWidget(*self*, *QWidget*) → QWidget

setButtonSymbols(*buttonSymbols*)

Set the buttonSymbols property, and update already existing widgets.

displayFormat()

Return the currently active display format string.

setDisplayFormat(*displayFormat*)

Set the display format string to use.

The default is "{num} of {total}".

pageCount()

Return the currently set page count.

setPageCount(*pageCount*)

Set the page count.

currentPageNumber()

Return the current page number.

setCurrentPageNumber(*num*)

Set our current page number.

updateCurrentPageNumber(*num*)

Set our current page number, but without emitting the signal.

class ZoomerAction(**args*, ***kwargs*)

Bases: PyQt5.QtWidgets.QWidgetAction

ZoomerAction provides a combobox with view modes and zoom factors.

zoomFactorChanged

viewModeChanged

viewModes()

Return the view modes that are displayed in the combobox.

See setViewModes() for explanation.

setViewModes(*modes*)

Set the view modes to display on top of the zoom values in the box.

An iterable of tuples (mode, name) is expected; every mode is a viewMode, the name is displayed. By default modes 1, 2 and 3 are displayed with the names "Width", "Height", "Page".

zoomFactors()

Return the zoom factors that are displayed in the combobox.

A zoom factor of 100% is represented by a floating point value of 1.0.

setZoomFactors(*factors*)

Set the zoom factors to display in the combobox.

A zoom factor of 100% is represented by a floating point value of 1.0.

zoomFormat()

Return the format string used to display zoom factors.

setZoomFormat(*zoomFormat*)

Set the format string used to display zoom factors.

createWidget(*self*, *QWidget*) → QWidget

viewMode()

Return the current view mode.

setViewMode(*mode*)

Set the current view mode.

zoomFactor()

Return the current zoom factor.

setZoomFactor(*factor*)

Set the current zoom factor.

setCurrentIndex(*index*)

Called when the user chooses an entry in a combobox.

2.5.31 The view module

The View, deriving from QAbstractScrollArea.

class Position(*pageNumber*, *x*, *y*)

Bases: `tuple`

pageNumber

Alias for field number 0

x

Alias for field number 1

y

Alias for field number 2

class View(*parent=None*, ***kwds*)

Bases: *qpageview.scrollarea.ScrollArea*

View is a generic scrollable widget to display Pages in a layout.

Using setPageLayout() you can set a PageLayout to the View, and you can add Pages to the layout using a list-like api. (PageLayout derives from list). A simple PageLayout is set by default. Call updatePageLayout() after every change to the layout (like adding or removing pages).

You can also add a Magnifier to magnify parts of a Page, and a Rubberband to enable selecting a rectangular region.

View emits the following signals:

pageCountChanged **(int)** emitted when the total amount of pages has changed

currentPageNumberChanged **(int)** emitted when the current page number has changed (starting with 1)

viewModeChanged **(int)** emitted when the `viewMode` has changed

rotationChanged **(int)** emitted when the `rotation` has changed

orientationChanged **(int)** emitted when the `orientation` has changed

zoomFactorChanged **(float)** emitted when the `zoomFactor` has changed

continuousModeChanged **(bool)** emitted when the `continuousMode` has changed

pageLayoutModeChanged **(str)** emitted when the `pageLayoutMode` has changed

pageLayoutUpdated () emitted whenever the page layout has been updated (redraw/resize)

MIN_ZOOM = 0.05

MAX_ZOOM = 64.0

wheelZoomingEnabled = True

whether to enable mouse wheel zooming

kineticPagingEnabled = True

whether to enable kinetic scrolling while paging (setCurrentPageNumber)

pagingOnScrollEnabled = True

whether to keep track of current page while scrolling

clickToSetCurrentPageEnabled = True

whether a mouse click in a page makes it the current page

strictPagingEnabled = False

whether PageUp and PageDown call setCurrentPageNumber instead of scroll

documentPropertyStore = None

can be set to a DocumentPropertyStore object. If set, the object is used to store certain View settings on a per-document basis. (This happens in the *clear()* and *setDocument()* methods.)

pageCountChanged

(int) emitted when the total amount of pages has changed

currentPageNumberChanged

(int) emitted when the current page number has changed (starting with 1)

viewModeChanged

(int) emitted when the `viewMode` has changed

rotationChanged

(int) emitted when the `rotation` has changed

orientationChanged

(int) emitted when the `orientation` has changed

zoomFactorChanged

(float) emitted when the `zoomFactor` has changed

continuousModeChanged

(bool) emitted when the `continuousMode` has changed

pageLayoutModeChanged

(str) emitted when the `pageLayoutMode` has changed

pageLayoutUpdated

emitted whenever the page layout has been updated (redraw/resize)

pageCount()

Return the number of pages in the view.

currentPageNumber()

Return the current page number in view (starting with 1).

setCurrentPageNumber(*num*)

Scrolls to the specified page number (starting with 1).

If the page is already in view, the view is not scrolled, otherwise the view is scrolled to center the page. (If the page is larger than the view, the top-left corner is positioned top-left in the view.)

updateCurrentPageNumber(*num*)

Set the current page number without scrolling the view.

gotoNextPage()

Convenience method to go to the next page.

gotoPreviousPage()

Convenience method to go to the previous page.

currentPage()

Return the page pointed to by currentPageNumber().

page(*num*)

Return the page at the specified number (starting at 1).

pages()

Return a list of all Pages in the page layout.

position()

Return a three-tuple Position(pageNumber, x, y).

The Position describes where the center of the viewport is on the layout. The page is the page number (starting with 1) and x and y the position on the page, in a 0..1 range. This way a position can be remembered even if the zoom or orientation of the layout changes.

setPosition(*position*, *allowKinetic=True*)

Centers the view on the spot stored in the specified Position.

If allowKinetic is False, immediately jumps to the position, otherwise scrolls smoothly (if kinetic scrolling is enabled).

setPageLayout(*layout*)

Set our current PageLayout instance.

The dpiX and dpiY attributes of the layout are set to the physical resolution of the widget, which should result in a natural size of 100% at zoom factor 1.0.

pageLayout()

Return our current PageLayout instance.

pageLayoutModes()

Return a dictionary mapping names to callables.

The callable returns a configured LayoutEngine that is set to the page layout. You can reimplement this method to returns more layout modes, but it is required that the name "single" exists.

pageLayoutMode()

Return the currently set page layout mode.

setPageLayoutMode(*mode*)

Set the page layout mode.

The mode is one of the names returned by pageLayoutModes(). The mode name "single" is guaranteed to exist.

updatePageLayout(*lazy=False*)

Update layout, adjust scrollbars, keep track of page count.

If lazy is set to True, calls lazyUpdate() to update the view.

modifyPages()

Return the list of pages and enter a context to make modifications.

Note that the first page is at index 0. On exit of the context the page layout is updated.

modifyPage(*num*)

Return the page (numbers start with 1) and enter a context.

On exit of the context, the page layout is updated.

clear()

Convenience method to clear the current layout.

setPages(*pages*)

Load the iterable of pages into the View.

Existing pages are removed, and the document is set to None.

setDocument(*document*)

Set the Document to display (see document.Document).

document()

Return the Document currently displayed (see document.Document).

reload()

If a Document was set, invalidate()s it and then reloads it.

loadPdf(*filename*, *renderer=None*)

Convenience method to load the specified PDF file.

The filename can also be a QByteArray or an already loaded popplerqt5.Poppler.Document instance.

loadSvgs(*filenames*, *renderer=None*)

Convenience method to load the specified list of SVG files.

Each SVG file is loaded in one Page. A filename can also be a QByteArray.

loadImages(*filenames*, *renderer=None*)

Convenience method to load images from the specified list of files.

Each image is loaded in one Page. A filename can also be a QByteArray or a QImage.

print(*printer=None*, *pageNumbers=None*, *showDialog=True*)

Print all, or speficied pages to QPrinter printer.

If given the pageNumbers should be a list containing page numbers starting with 1. If showDialog is True, a print dialog is shown, and printing is canceled when the user cancels the dialog.

If the QPrinter to use is not specified, a default one is created. The print job is started and returned (a printing.PrintJob instance), so signals for monitoring the progress could be connected to. (If the user cancels the dialog, no print job is returned.)

static properties()

Return an uninitialized ViewProperties object.

readProperties(*settings*)

Read View settings from the QSettings object.

If a documentPropertyStore is set, the settings are also set as default for the DocumentPropertyStore.

writeProperties(*settings*)

Write the current View settings to the QSettings object.

If a documentPropertyStore is set, the settings are also set as default for the DocumentPropertyStore.

setViewMode(*mode*)

Sets the current ViewMode.

viewMode()

Returns the current ViewMode.

setRotation(*rotation*)

Set the current rotation.

rotation()

Return the current rotation.

rotateLeft()

Rotate the pages 270 degrees.

rotateRight()

Rotate the pages 90 degrees.

setOrientation(*orientation*)

Set the orientation (Horizontal or Vertical).

orientation()

Return the current orientation (Horizontal or Vertical).

setContinuousMode(*continuous*)

Sets whether the layout should display all pages.

If True, the layout shows all pages. If False, only the page set containing the current page is displayed. If the pageLayout() does not support the PageSetLayoutMixin methods, this method does nothing.

continuousMode()

Return True if the layout displays all pages.

displayPageSet(*what*)

Try to display a page set (if the layout is not in continuous mode).

what can be:

"next": go to the next page set "previous": go to the previous page set "first": go to the first page set "last": go to the last page set integer: go to the specified page set

setMagnifier(*magnifier*)

Sets the Magnifier to use (or None to disable the magnifier).

The viewport takes ownership of the Magnifier.

magnifier()

Returns the currently set magnifier.

setRubberband(*rubberband*)

Sets the Rubberband to use for selections (or None to not use one).

rubberband()

Return the currently set rubberband.

pagingOnScrollDisabled()

During this context a scroll is not tracked to update the current page number.

scrollContentsBy(*dx*, *dy*)

Reimplemented to move the rubberband and adjust the mouse cursor.

stopScrolling()

Reimplemented to adjust the mouse cursor on scroll stop.

fitPageLayout()

Fit the layout according to the view mode.

Does nothing in FixedScale mode. Prevents scrollbar/resize loops by precalculating which scrollbars will appear.

keepCentered(*pos=None*)

Context manager to keep the same spot centered while changing the layout.

If pos is not given, the viewport's center is used. After yielding, updatePageLayout() is called.

setZoomFactor(*factor*, *pos=None*)

Set the zoom factor (1.0 by default).

If pos is given, that position (in viewport coordinates) is kept in the center if possible. If None, zooming centers around the viewport center.

zoomFactor()

Return the page layout's zoom factor.

zoomIn(*pos=None*, *factor=1.1*)

Zoom in.

If pos is given, it is the position in the viewport to keep centered. Otherwise zooming centers around the viewport center.

zoomOut(*pos=None*, *factor=1.1*)

Zoom out.

If pos is given, it is the position in the viewport to keep centered. Otherwise zooming centers around the viewport center.

zoomNaturalSize(*pos=None*)

Zoom to the natural pixel size of the current page.

The natural pixel size zoom factor can be different than 1.0, if the screen's DPI differs from the current page's DPI.

layoutPosition()

Return the position of the PageLayout relative to the viewport.

This is the top-left position of the layout, relative to the top-left position of the viewport.

If the layout is smaller than the viewport it is centered by default. (See ScrollArea.alignment.)

visibleRect()

Return the QRect of the page layout that is currently visible in the viewport.

visiblePages(*rect=None*)

Yield the Page instances that are currently visible.

If rect is not given, the visibleRect() is used. The pages are sorted so that the pages with the largest visible part come first.

ensureVisible(*rect*, *margins=None*, *allowKinetic=True*)

Ensure rect is visible, switching page set if necessary.

adjustCursor(*pos*)

Sets the correct mouse cursor for the position on the page.

repaintPage(*page*)

Call this when you want to redraw the specified page.

lazyUpdate(*page=None*)

Lazily repaint page (if visible) or all visible pages.

Defers updating the viewport for a page until all rendering tasks for that page have finished. This reduces flicker.

rerender(*page=None*)

Schedule the specified page or all pages for rerendering.

Call this when you have changed render options or page contents. Repaints the page or visible pages lazily, reducing flicker.

pagesToPaint(*rect*, *painter*)

Yield (page, rect) to paint in the specified rectangle.

The specified rect is in viewport coordinates, as in the paint event. The returned rect describes the part of the page actually to draw, in page coordinates. (The full rect can be found in page.rect().) Translates the painter to the top left of each page.

The pages are sorted with largest area last.

event(*ev*)

Reimplemented to get Gesture events.

handleGestureEvent(*event*)

Gesture event handler.

Return False if event is not accepted. Currently only cares about PinchGesture. Could also handle Swipe and Pan gestures.

pinchGesture(*gesture*)

Pinch gesture event handler.

Return False if event is not accepted. Currently only cares about ScaleFactorChanged and not RotationAngleChanged.

paintEvent(*ev*)

Paint the contents of the viewport.

resizeEvent(*ev*)

Reimplemented to scale the view if needed and update the scrollbars.

wheelEvent(*ev*)

Reimplemented to support wheel zooming and paging through page sets.

mousePressEvent(*ev*)

Implemented to set the clicked page as current, without moving it.

mouseMoveEvent(*ev*)

Implemented to adjust the mouse cursor depending on the page contents.

keyPressEvent(*ev*)

Reimplemented to go to next or previous page set if possible.

class ViewProperties

Bases: `object`

Simple helper class encapsulating certain settings of a View.

The settings can be set to and got from a View, and saved to or loaded from a QSettings group.

Class attributes serve as default values, None means: no change. All methods return self, so operations can easily be chained.

If you inherit from a View and add more settings, you can also add properties to this class by inheriting from it. Reimplement View.properties() to return an instance of your new ViewProperties subclass.

position = None

rotation = 0

zoomFactor = 1.0

viewMode = 0

orientation = None

continuousMode = None

pageLayoutMode = None

setdefaults()

Set all properties to default values. Also used by View on init.

copy()

Return a copy or ourselves.

names()

Return a tuple with all the property names we support.

mask(*names*)

Set properties not listed in names to None.

get(*view*)

Get the properties of a View.

set(*view*)

Set all our properties that are not None to a View.

save(*settings*)

Save the properties that are not None to a QSettings group.

load(*settings*)

Load the properties from a QSettings group.

class DocumentPropertyStore

Bases: `object`

Store ViewProperties (settings) on a per-Document basis.

If you create a DocumentPropertyStore and install it in the documentPropertyStore attribute of a View, the View will automatically remember its settings for earlier displayed Document instances.

default = None

mask = None

get(*document*)

Get the View properties stored for the document, if available.

If a ViewProperties instance is stored in the *default* attribute, it is returned when no properties were available. Otherwise, None is returned.

set(*document*, *properties*)

Store the View properties for the document.

If the *mask* attribute is set to a list or tuple of names, only the listed properties are remembered.

2.5.32 The widgetoverlay module

View mixin class to display QWidgets on top of a Page.

class OverlayData(*page*, *point*, *rect*, *alignment*)

Bases: `tuple`

alignment

Alias for field number 3

page

Alias for field number 0

point

Alias for field number 1

rect

Alias for field number 2

class WidgetOverlayViewMixin(*parent=None*)

Bases: object

Mixin class to add widgets to be displayed on top of pages.

Widgets are added using addWidget(), and become children of the viewport.

This class adds the following instance attribute:

deleteUnusedOverlayWidgets = True

If True, unused widgets are deleted using QObject.deleteLater(). Otherwise, only the parent is set to None. A widget becomes unused if the Page it was added to disappears from the page layout.

deleteUnusedOverlayWidgets = True

addWidget(*widget*, *page*, *where=None*, *alignment=None*)

Add widget to be displayed on top of page.

The widget becomes a child of the viewport.

The *where* argument can be a QPoint or a QRect. If a rect is given, the widget is resized to occupy that rectangle. The rect should be in page coordinates. When the zoom factor is changed, the widget will be resized.

If a point is given, the widget is not resized and aligned on the point using the specified alignment (top-left if None).

If where is None, the widget occupies the whole page.

You can also use this method to change the page or rect for a widget that already has been added.

removeWidget(*widget*)

Remove the widget.

The widget is not deleted, but its parent is set to None.

widgets(*page=None*)

Yield all widgets (for the Page if given).

removeWidgets(*page=None*)

Remove all widgets (for the Page if given).

The widget are not deleted, but their parent is set to None.

updatePageLayout(*lazy=False*)

Reimplemented to update the size and position of the widgets.

scrollContentsBy(*dx*, *dy*)

Reimplemented to scroll the page widgets along with the layout.

resizeEvent(*ev*)

Reimplemented to keep page widgets in the right position.

2.6 Installing qpageview

This package installs one Python package, `qpageview`, in the usual location for Python modules.

You can install qpageview without downloading it first via `pip`:

```
pip install qpageview
```

You can also install from the source directory:

```
python3 setup.py install
```

2.7 ChangeLog

2022-05-05: qpageview-0.6.2

- Maintainance release
- Kept another implicit float->int conversion from happening by having Scrollarea.remainingScrollTime() returning an int
- Some robustness improvements
- Documentation improvements

2021-11-11: qpageview 0.6.1

- View.strictPagingEnabled always lets PgUp/PgDn scroll a page instead of a screenful
- Don't depend on implicit float->int conversions, which were deprecated since Python 3.8 and not supported anymore by Python 3.10
- Fixed initial zoomfactor for ImageView when fitNaturalSizeEnabled is True

2021-01-07: qpageview 0.6.0

- added view.View.pages() method (#2)
- added view.View.setPages() method (inspired by #4)

2020-04-25: qpageview 0.5.1

- Many documentation updates
- Add PagerAction.setButtonSymbols()
- fix flickering mouse cursor on rubberband
- make it easier to manipulate the edge/corner of the rubberband

2020-04-19: qpageview 0.5.0

Initial release. The qpageview module was developed by me, Wilbert Berendsen, as a replacement of the qpopplerview module inside Frescobaldi, the LilyPond sheet music text editor. I decided that it would be best if qpageview became its own project, to make it easier to use this package in other applications.

2.8 License

The *qpageview* package is licensed under the General Public License v3.

2.8.1 GNU General Public License

Version 3, 29 June 2007 Copyright © 2007 Free Software Foundation, Inc <http://fsf.org>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: **(1)** assert copyright on the software, and **(2)** offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

2.8.2 0. Definitions

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that **(1)** displays an appropriate copyright notice, and **(2)** tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

2.8.3 1. Source Code

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that **(a)** is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and **(b)** serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2.8.4 2. Basic Permissions

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

2.8.5 3. Protecting Users' Legal Rights From Anti-Circumvention Law

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

2.8.6 4. Conveying Verbatim Copies

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

2.8.7 5. Conveying Modified Source Versions

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- **a)** The work must carry prominent notices stating that you modified it, and giving a relevant date.
- **b)** The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- **c)** You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

- **d)** If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

2.8.8 6. Conveying Non-Source Forms

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- **a)** Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- **b)** Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either **(1)** a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or **(2)** access to copy the Corresponding Source from a network server at no charge.
- **c)** Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- **d)** Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- **e)** Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either **(1)** a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or **(2)** anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

2.8.9 7. Additional Terms

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- **a)** Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- **b)** Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- **c)** Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- **d)** Limiting the use for publicity purposes of names of licensors or authors of the material; or
- **e)** Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- **f)** Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

2.8.10 8. Termination

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated **(a)** provisionally, unless and until the copyright holder explicitly and finally terminates your license, and **(b)** permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

2.8.11 9. Acceptance Not Required for Having Copies

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

2.8.12 10. Automatic Licensing of Downstream Recipients

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

2.8.13 11. Patents

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either **(1)** cause the Corresponding Source to be so available, or **(2)** arrange to deprive yourself of the benefit of the patent license for this particular work, or **(3)** arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license **(a)** in connection with copies of the covered work conveyed by you (or copies made from those copies), or **(b)** primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

2.8.14 12. No Surrender of Others' Freedom

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

2.8.15 13. Use with the GNU Affero General Public License

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

2.8.16 14. Revised Versions of this License

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

2.8.17 15. Disclaimer of Warranty

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

2.8.18 16. Limitation of Liability

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

2.8.19 17. Interpretation of Sections 15 and 16

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands *show w* and *show c* should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

q

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z