`pageturner.py`: A Tkinter widget for turning pages

John W. Shipman

2009-10-10 13:15

Abstract

Describes a graphical user interface widget that allows the selection of one of a set of pages, using the Python programming language and the Tkinter widget set.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **`tcc-doc@nmt.edu`**.

Table of Contents

1. What does the `pageturner.py` widget do? 1
2. General layout 2
3. How to use a `PageTurner` widget 3
4. Design decisions 3
    4.1. How it works 3
    4.2. Limitations 4
5. Prologue to the `pageturner.py` module 4
6. **`class PageTurner`** 4
7. **`PageTurner.__init__()`**: The constructor 6
8. **`PageTurner.__createWidgets()`**: Place internal widgets 6
9. **`PageTurner.__createButtons()`**: Create the controls 8
10. **`PageTurner.__pageNoHandler()`**: Jump to a user-selected page 10
11. **`PageTurner.__prevHandler()`**: Go to previous page 11
12. **`PageTurner.__nextHandler()`**: Go to next page 12
13. **`PageTurner.addPage()`**: Add a new content page 12
14. **`PageTurner.setPageNo()`**: Select which page is displayed 12
15. A small test driver 14

1. What does the `pageturner.py` widget do?

This document describes a widget that can be included in a GUI (graphical user interface) application to present one of a set of multiple pages in a frame. The user is presented with “Next” and “Previous” buttons that allow them to page forward and backward through the page set. Pages can also be changed under program control.

This document assumes that the reader is familiar with these tools:

[1] http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/
[2] http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/pageturner.pdf

- The application is written in Python, the author's favorite programming language. For more information, see *Python 2.2 quick reference*[3].
- Tkinter is a graphical user interface for Python; for more information see *Tkinter reference: A GUI for Python*[4].

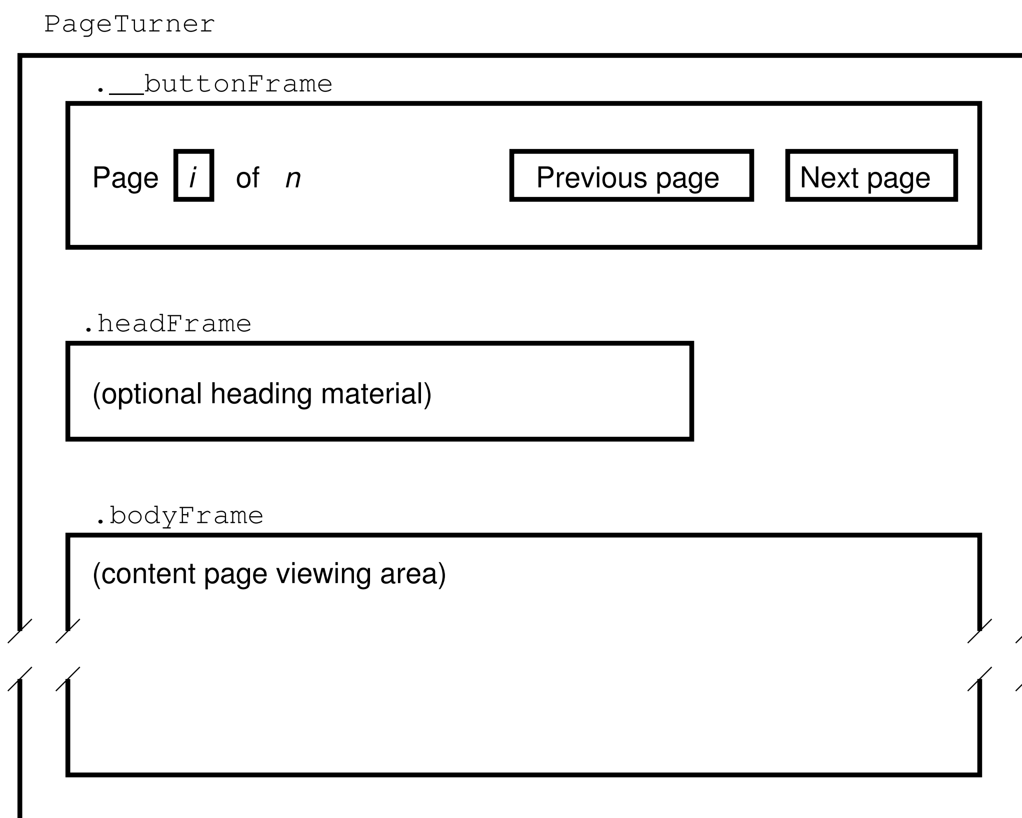
This document describes the external interface to the `pageturner.py` widget. It also contains the code for the widget, in literate programming style: for more on literate programming, see *A source extractor for lightweight literate programming*[5]. The code was developed using the Cleanroom or zero-defect style; for the author's practice and conventions, see *The cleanroom software development methodology*[6].

Relevant online files:

- The `pageturner.py` module itself.[7]
- The `pageturnertest` test driver.[8]

2. General layout

This widget is basically a container class for a list of frames, called *content pages*, along with buttons that allow the user to go to the next or previous frame. On the screen it looks like this:



[3] http://www.nmt.edu/tcc/help/pubs/python22
[4] http://www.nmt.edu/tcc/help/pubs/tkinter
[5] http://www.nmt.edu/tcc/help/lang/python/examples/litsource
[6] http://www.nmt.edu/~shipman/soft/clean/
[7] http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/pageturner.py
[8] http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/pageturnertest

From top to bottom, a `PageTurner` widget contains:

- A private frame, the *button frame*, containing the operating controls:
  - A label of the form "Page *i* of *n*", showing the user where they are in the page set. The external page number counts from 1, showing 0 only when there are no content pages.

    The user can also edit the current page number (*i* in the diagram) to jump to a specific page.
  - A *Previous Page* button that changes the page viewing area to the previous page, if any.
  - A *Next Page* button that changes the page viewing area to the next page in sequence, if any.
- An exported frame, the **`.headFrame`** attribute of the `PageTurner` widget, where you can place any titles or other material you want to appear above the page viewing area.
- An exported frame, the **`.bodyFrame`** attribute of the `PageTurner` widget, which holds the content pages to be viewed. You can make this frame fixed in size or allow it to resize for each displayed content page.

3. How to use a `PageTurner` widget

To add a `PageTurner` widget to your Tkinter application, use this procedure:

1. Instantiate the widget in some parent window, and grid it. Let's call the instance ***PT***. Here's the general form of the constructor:

   ```
   PT = PageTurner ( master=None, size=None )
   PT.grid(...)
   ```

   master is the parent widget in which you want to create the `PageTurner` widget. If you omit the **`size`** argument, the page viewing area will be resized for each page as it is displayed. If you want the page viewing area to have a minimum size, use an argument of the form **`size=(`*`w`*`, `*`h`*`)`** where ***w*** is the width and ***h*** the width of the area, expressed in pixels.
2. If you want to display anything between the button frame and the page viewing area, place the material into the **`Frame`** widget ***PT*`.headFrame`**.
3. Create **`Frame`** widgets for each content page; each such widget should have ***Pt*** as its parent. Then for each content page frame ***f***, call the ***PT*`.addPage(`*`f`*`)`** method to add the frame as the next page.

   Do not grid these widgets as you create them. Until the next step, nothing will appear in the page viewing area.
4. Use the method ***PT*`.setPageNo(`*`n`*`)`** to display page ***n***, counting from 1. The default value for ***n*** is 1, the first content page.

4. Design decisions

4.1. How it works

Every widget (except the root window) has a parent. However, the child widget does not actually appear on the screen until it is gridded with the **`.grid()`** method.

So, to give the illusion that many different pages can appear in the same space, each page inside a PageTurner is a single child widget of its **.bodyFrame** attribute, but only one is gridded at any given time.

The first time a new page widget is added to a PageTurner by calling its **.addPage()** method, the new page is gridded inside the **.bodyFrame** at row 0, column 0, at which point it appears on the screen.

From then on, whenever we want to present a different page, we can make the current page invisible by calling its **.grid_forget()** method. This removes that child from the screen, but it is still registered as a child of **.bodyFrame**, and can be put back anytime by calling its **.grid()** method. (In this respect **.grid_forget()** differs from **.grid_remove()**, which destroys the child widget altogether.)

4.2. Limitations

This widget is intended for applications where the set of content pages is static, or at least grows only at the end. If there is a need for a widget to display a dynamically changing set of pages, new external methods will be required to support page insertion and deletion at arbitrary locations in the list of content pages. A particularly Pythonic way to do that might be to mimic the methods on list types such as **.insert()**, **.append()**, and the **del** statement.

In any event, the addition of such features will require a serious rethinking of the internal attributes that manage the content pages.

5. Prologue to the pageturner.py module

The balance of this document describes the actual code of the pageturner.py widget.

First there is a brief opening comment, the usual Python documentation string for the module:

pageturner.py

```
"""pageturner.py: Tkinter widget for viewing a sequence of content pages
  Do not edit this file.  It is extracted automatically from its
  documentation file.  See:
    http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/
"""
```

Next is the obligatory importation of the **Tkinter** module.

pageturner.py

```
#================================================================
# Imports
#----------------------------------------------------------------

from Tkinter import *
```

6. class PageTurner

Here is the start of the actual widget class, and the formal declaration of its exported and private attributes and methods.

pageturner.py

```

# - - - - -   c l a s s   P a g e T u r n e r   - - - - -

```

```
class PageTurner(Frame):
    """Widget to allow user to page through a set of content pages.

      Exports:
        PageTurner ( master=None, size=None ):
          [ (master is the containing widget) and
            (size is a (width, height) tuple to get a fixed
            page viewing area of that size, or None for one
            that automatically resizes for each page) ->
              return a new PageTurner widget with those attributes
              and no content pages ]
        .pageNoEntry:       [ displays the current page number ]
        .nPagesLabel:       [ displays the number of pages ]
        .prevButton:        [ the 'Previous page' button ]
        .nextButton:        [ the 'Next page' button ]
        .headFrame:
          [ a Frame inside self for displaying page headings, if any ]
        .bodyFrame:
          [ a Frame inside self for displaying the current
            content page, if any ]
        .addPage ( f ):
          [ f is a Frame ->
              self  :=  self with f added as its next content page ]
        .setPageNo ( n ):
          [ n is a positive integer ->
             if n < (number of content pages) ->
               self  :=  self displaying page n (counting from 1)
             else -> I ]
```

Having described the exported interface, the balance of the documentation string for the class is a list of the contained widgets and other miscellaneous private attributes.

pageturner.py

```
      Included widgets:
        .__buttonFrame:     [ frame containing all operating controls ]

      Invariants:
        .__size:
          [ the size argument passed to the constructor ]
        .__pageList:
          [ a list whose members are the content pages in viewing
            order ]
        .__pageNo:
           [ if self contains any pages ->
               current page number, counting from 1
             else -> 0 ]
        .__pageNoVar:
          [ the StringVar associated with self.pageNoEntry.  Invariant:
            if len(self.__pageList) == 0 -> 0
            else -> self.__pageNo ]
        .__nPagesVar:
          [ the StringVar associated with self.nPagesLabel ]
    """
```

All the internal control widgets such as **.nextButton**, the “Next page” button, are exported so that you can configure such attributes as their fonts and colors by using the universal Tkinter widget method **.configure()**. Obviously this gives the caller the ability to break encapsulation rather heavily, so it would be prudent to change only superficial attributes of the control (such as **background**) and not critical operational attriubutes like **command**.

Here's the way the widgets are gridded in a `PageTurner` widget:

	0
0	.__buttonFrame
1	.headFrame
2	.bodyFrame

For the gridding of the controls inside the **.__buttonFrame**, see Section 9, “**PageTurner.__createButtons()**: Create the controls” (p. 8).

7. PageTurner.__init__(): The constructor

The constructor has four general duties: call the parent class constructor; process the arguments; set up invariants on the attributes; and create the widgets. The latter is delegated to the **.__createWidgets()** method.

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ i n i t _ _   - - -

    def __init__ ( self, master=None, size=None ):
        """Constructor for the PageTurner widget."""

        #-- 1 --
        # [ master  :=  master with a new Frame added but not gridded
        #   self    :=  that new Frame ]
        Frame.__init__ ( self, master )

        #-- 2 --
        self.__size  =  size

        #-- 3 --
        self.__pageList  =  []
        self.__pageNo    =  0

        #-- 4 --
        # [ self  :=  self with all widgets created ]
        self.__createWidgets()
```

8. PageTurner.__createWidgets(): Place internal widgets

This method creates and grids all the widgets inside a `PageTurner`. Its steps mirror the three major components: the controls (**self.__buttonFrame**), the heading frame (**self.headFrame**), and the body frame (**self.bodyFrame**).

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ c r e a t e W i d g e t s   - - -

    def __createWidgets ( self ):
        """Create and grid all internal widgets.

          [ self.__size is as invariant ->
              self  :=  self with all widgets created ]
        """
```

The **.__buttonFrame** is gridded with **sticky=E+W** so that it will be stretched across the full width of the `PageTurner`. This will allow us to place the *Next page* button at the top right corner.

pageturner.py

```
        #-- 1 --
        # [ self  :=  self with a new Frame added and gridded
        #             containing self's control widgets ]
        self.__buttonFrame  =  self.__createButtons()
        rowx  =  0
        self.__buttonFrame.grid ( row=rowx, column=0,
            columnspan=99, sticky=E+W )
```

Next we add empty frames for the heading and for the content page display area.

pageturner.py

```
        #-- 2 --
        # [ self            :=  self with a new, empty Frame added
        #                       and gridded
        #   self.headFrame  :=  that Frame ]
        self.headFrame  =  Frame ( self )
        rowx  +=  1
        self.headFrame.grid ( row=rowx, sticky=W )

        #-- 3 --
        # [ self            :=  self with a new, empty Frame added
        #                       and gridded
        #   self.bodyFrame  :=  that Frame ]
        self.bodyFrame  =  Frame ( self, relief=SUNKEN,
                                   borderwidth=2 )
        rowx  +=  1
        self.bodyFrame.grid ( row=rowx, sticky=W )
```

One obscure operation remains. In general, the **.grid()** geometry manager allows all widgets to resize dynamically to fit their contents. However, if the user has provided a **size** argument to the constructor, they want the **self.bodyFrame** to be fixed in size. To do this, it is necessary to disable "propagation" of internal dimensions to the containing widget. The author spent a lot of time finding the right universal method, **.grid_propagate()**[9].

pageturner.py

```
        #-- 4 --
        # [ if self.__size is not None ->
        #     self.bodyFrame  :=  self.bodyFrame fixed as size
        #                         self.__size
        #   else -> I ]
```

[9] http://www.nmt.edu/tcc/help/pubs/tkinter/universal.html

```
        if  self.__size:
            self.bodyFrame['width']   =  self.__size[0]
            self.bodyFrame['height']  =  self.__size[1]
            self.bodyFrame.grid_propagate(0)
```

9. PageTurner.__createButtons(): Create the controls

This method creates a **Frame** containing all the controls, and returns that frame to the caller for gridding.

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ c r e a t e B u t t o n s   - - -

    def __createButtons ( self ):
        """Create the frame containing all the controls.

          [ return a new Frame containing self's control widgets ]
        """
```

All the widgets in the control frame are arranged in one horizontal row. However, we want some of them all the way to the left of the frame and some all the way to the right. This cannot be done just by using **sticky=W** on the first group and **sticky=E** for the rest: the **sticky** attribute affects only where widgets are placed within their columns, but it does not make the columns stretchable.

In order to make columns expandable, it is necessary to call the **.columnconfigure()** method on the parent widget and use a **weight=*N*** argument to specify where the extra space will be distributed. To get the positioning we want, we insert a dummy column between the left-hand and right-hand groups of controls, assign it **weight=10000** with the **.columnconfigure** method, and give the other columns **weight=1**.

Here, then, is a list of the columns gridded inside the control frame, and their contents.

Column	Attribute	Purpose
0	**.__pageLabel**	The word "Page".
1	**.pageNoEntry**	**Entry** widget to display the current page number and allow the user to modify it. The associated control variable is **.__pageNoVar**.
2	**.__ofLabel**	The word "of".
3	**.nPagesLabel**	Displays the total number of pages. The associated control variable is **.__nPagesVar**.
4	—	Dummy column to take up the space between the left-hand and right-hand control groups.
5	**.prevButton**	The "Previous page" button.
6	**.nextButton**	The "Next page" button.

First we create the frame to be returned to the caller.

pageturner.py

```
        #-- 1 --
        # [ self     :=  self with a new Frame added
        #   buttons  :=  that new Frame ]
        buttons  =  Frame ( self )
```

In order for the **.pageNoEntry** widget to respond to a user's changing the page number, it must have an event binding for the *Enter* key, which is a **"<KeyPress-Return>"** event. For the handler that is invoked when the user edits **self.pageNoEntry**, see Section 10, "**PageTurner.__pageNoHandler()**: Jump to a user-selected page" (p. 10).

pageturner.py

```
        #-- 2 --
        # [ buttons  :=  buttons with a Label added containing "Page " ]
        self.__pageLabel  =  Label ( buttons, text="Page " )
        colx  =  0
        self.__pageLabel.grid ( row=0, column=colx )
        buttons.columnconfigure(colx, weight=1)

        #-- 3 --
        # [ buttons  :=  buttons with an Entry added with a StringVar
        #       control variable, and a handler that turns to the
        #       given page when the user presses Enter
        #   self.pageNoEntry  :=  that Entry
        #   self.__pageNoVar  :=  that StringVar
        self.__pageNoVar  =  StringVar()
        self.__pageNoVar.set("0")
        self.pageNoEntry  =  Entry ( buttons, relief=SUNKEN, width=3,
            textvariable=self.__pageNoVar )
        self.pageNoEntry.bind ( "<KeyPress-Return>",
                                self.__pageNoHandler )
        colx  +=  1
        self.pageNoEntry.grid ( row=0, column=colx )
        buttons.columnconfigure(colx, weight=1)

        #-- 4 --
        # [ buttons  :=  buttons with a Label added containing " of " ]
        self.__ofLabel  =  Label ( buttons, text=" of " )
        colx  +=  1
        self.__ofLabel.grid ( row=0, column=colx )
        buttons.columnconfigure(colx, weight=1)

        #-- 5 --
        # [ buttons  :=  buttons with a Label added with a StringVar
        #             control variable
        #   self.nPagesLabel  :=  that Label
        #   self.__nPagesVar  :=  that StringVar ]
        self.__nPagesVar  =  StringVar()
        self.__nPagesVar.set("0")
        self.nPagesLabel  =  Label ( buttons, relief=FLAT, width=3,
            textvariable=self.__nPagesVar )
        colx  +=  1
        self.nPagesLabel.grid ( row=0, column=colx )
        buttons.columnconfigure(colx, weight=1)
```

At this point we allocate the dummy column and give it a huge **weight** configuration so it will absorb the remaining space.

pageturner.py

```
        #-- 6 --
        # [ buttons  :=  buttons with an empty column that
```

```
        #                 expands by a factor of 10000 ]
        colx  +=  1
        buttons.columnconfigure(colx, weight=10000)
```

The “Previous page” and “Next page” buttons are both initially disabled; they will be enabled when the first content page is added. For the button handlers, see Section 11, “**PageTurner.__prevHandler()**: Go to previous page” (p. 11) and Section 12, “**PageTurner.__nextHandler()**: Go to next page” (p. 12).

pageturner.py

```
        #-- 7 --
        # [ buttons  :=  buttons with a new Button added that moves to
        #              the previous page
        #   self.prevButton  :=  that Button ]
        self.prevButton  =  Button ( buttons, text="Previous page",
            command=self.__prevHandler, state=DISABLED )
        colx  +=  1
        self.prevButton.grid ( row=0, column=colx, sticky=E )
        buttons.columnconfigure(colx, weight=1)

        #-- 8 --
        # [ buttons  :=  buttons with a new Button added that moves to
        #                 the next page
        #   self.nextButton  :=  that Button ]
        self.nextButton  =  Button ( buttons, text="Next page",
            command=self.__nextHandler, state=DISABLED )
        colx  +=  1
        self.nextButton.grid ( row=0, column=colx, sticky=E )
        buttons.columnconfigure(colx, weight=1)
```

All that remains is to return the completed frame to the caller.

pageturner.py

```
        #-- 9 --
        return buttons
```

10. PageTurner.__pageNoHandler(): Jump to a user-selected page

When the user presses the *Enter* key in the **self.pageNoEntry** field, this handler is called. If the user has entered a valid integer that is within the range of external page numbers (counting from 1), we call the **self.setPageNo()** method to present that page.

Because this handler uses the event binding (**.bind()**) rather than the command binding (**command=...)**, it is called with an **Event** object as an argument. However, we don't need the event to find out what happened. We simply inspect the value of the **.__pageNoVar** control variable and attempt to use it to change the page number.

If the user enters something other than a number in the **Entry** widget, or a number that is outside the range of page numbers, we could give them an error popup, but it should be sufficient feedback just to change the field's value back to the current page number.

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ p a g e N o H a n d l e r   - - -
```

```
    def __pageNoHandler ( self, event ):
        """Handle a change to self.pageNoEntry

          [ event is an Event object ->
              if self.__pageNoVar contains a valid integer that
              is in the range [1,len(self.__pageList)] ->
                self  :=  self displaying int(self.__pageNoVar.get())
              else ->
                self.pageNoEntry  :=  self.__pageNo as a string ]
        """
```

First we get the field value, attempt to convert it to an integer, and check that it is a valid external page number. If any of these steps fail, reset the page number and exit.

pageturner.py

```
        #-- 1 --
        # [ if self.__pageNoVar contains an integer in
        #   [1,len(self.__pageList)] ->
        #     self  :=  self displaying page (pageNo)
        #     return
        #   else -> I ]
        try:
            pageNo  =  int ( self.__pageNoVar.get() )
            if  ( 1 <= pageNo <= len(self.__pageList) ):
                self.setPageNo ( pageNo )
                return
        except ValueError:
            pass

        #-- 2 --
        # [ self.pageNoEntry  :=  self.__pageNo as a string ]
        self.__pageNoVar.set ( self.__pageNo )
```

11. `PageTurner.__prevHandler()`: Go to previous page

This handler is called when the user presses the “Previous page” button. Because it is connected to the widget using the “**`command=`**” argument to the **`Button`** constructor, it is not passed an **`Event`** object like the **`.__pageNoHandler()`** method.

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ p r e v H a n d l e r   - - -

    def __prevHandler ( self ):
        """Handle the 'Previous page' button.

          [ if self.__pageNo > 1 ->
              self  :=  self displaying page (self.__pageNo-1)
            else -> I ]
        """
        if  self.__pageNo > 1:
            self.setPageNo ( self.__pageNo - 1 )
```

12. PageTurner.__nextHandler(): Go to next page

This handler is called when the user presses the "Next page" button.

pageturner.py

```
# - - -   P a g e T u r n e r . _ _ n e x t H a n d l e r   - - -

    def __nextHandler ( self ):
        """Handle the 'Next page' button.

          [ if self.__pageNo < len(self.__pageList) ->
              self  :=  self displaying page (self.__pageNo+1)
            else -> I ]
        """
        if  self.__pageNo < len(self.__pageList):
            self.setPageNo ( self.__pageNo + 1 )
```

13. PageTurner.addPage(): Add a new content page

The caller has created a **Frame** widget and wants us to add it to the list of content pages. First we add it to **self.__pageList**.

pageturner.py

```
# - - -   P a g e T u r n e r . a d d P a g e   - - -

    def addPage ( self, page ):
        """Add a new content page."""

        #-- 1 --
        # [ self.__pageList   +:=  page
        #   self.nPagesLabel  :=   1+len(self.__pageList) ]
        self.__pageList.append ( page )
        self.__nPagesVar.set ( str ( len(self.__pageList) ) )
```

If this is the first page to be added, we also call the **.setPageNo()** method so that the page gets displayed. This method also sets the page number, **self.__pageNo**.

pageturner.py

```
        #-- 2 --
        # [ if self.__pageNo == 0 ->
        #     self  :=  self with the first page displayed
        #   else -> I ]
        if  self.__pageNo == 0:
            self.setPageNo ( 1 )
```

14. PageTurner.setPageNo(): Select which page is displayed

This method selects which content page is displayed in **self.bodyFrame**.

pageturner.py

```
# - - -   P a g e T u r n e r . s e t P a g e N o   - - -
```

```
    def setPageNo ( self, n ):
        """Display the nth page, counting from 1."""
```

First we test the value of **n** to be sure it is a valid page number. If not, we silently return.

pageturner.py

```

        #-- 1 --
        # [ if  (n < 1) or (n > len(self.__pageList)) ->
        #     return
        #   else -> I ]
        if  not ( 1 <= n <= len(self.__pageList) ):
            return
```

The invariant for **self.__pageNo** states that it is zero if there are no pages; otherwise it is the number of the current page, counting from 1. So, if **self.__pageNo** is greater than zero, we must remove the current page before we can display a new one.

pageturner.py

```
        #-- 2 --
        # [ if self.__pageNo > 0 ->
        #     self.__pageList[self.__pageNo-1]  :=  itself ungridded
        #   else -> I ]
        if  self.__pageNo > 0:
            oldPage  =  self.__pageList[self.__pageNo-1]
            oldPage.grid_forget()
```

Next we adjust the controls to reflect the newly selected page number. In addition to setting the current page number, we also set the total number of pages, because this might be the first time we're displaying a page.

pageturner.py

```
        #-- 3 --
        # [ self.__pageNoVar  :=  n
        #   self.__nPagesVar  :=  len(self.__pageList) ]
        self.__pageNoVar.set ( str ( n ) )
        self.__nPagesVar.set ( str ( len ( self.__pageList ) ) )
```

To make the new page appear, we grid it into **self.bodyFrame**. The **sticky** argument positions the content in the upper left corner of the frame. The call to the **.columnconfigure()** and **.rowconfigure()** methods on the parent widget make the column and row stretchable. Also, we store the new page number in **self.__pageNo**.

pageturner.py

```
        #-- 4 --
        # [ self.__pageList[n-1]  :=  itself, gridded
        #   self.__pageNo         :=  n ]
        newPage  =  self.__pageList[n-1]
        newPage.grid ( row=0, column=0, sticky=NW )
        self.bodyFrame.columnconfigure(0, weight=1)
        self.bodyFrame.rowconfigure(0, weight=1)
        self.__pageNo  =  n
```

There is one additional detail. The Next and Previous buttons are initially disabled. If we have gotten this far, we must also enable them.

pageturner.py

```
        #-- 5 --
        # [ self.prevButton  :=  self.prevButton enabled
        # [ self.nextButton  :=  self.nextButton enabled ]
        self.prevButton["state"]  =  NORMAL
        self.nextButton["state"]  =  NORMAL
```

15. A small test driver

Here is a small application to demonstrate the `PageTurner` widget. It has only two widgets: a `PageTurner` and a Quit button.

pageturnertest

```
#!/usr/bin/env python
#================================================================
# pageturnertest:  Test driver for the PageTurner widget.
#   Do not edit this file.  It is extracted automatically from its
#   documentation file.  See:
#     http://www.nmt.edu/tcc/help/lang/python/examples/pageturner/
#----------------------------------------------------------------
```

We import the **`Tkinter`** page as usual, then import all the functions of the widget under test.

pageturnertest

```
from Tkinter import *
from pageturner import *
```

The **`Application`** class is the base class for the GUI application.

pageturnertest

```

# - - - - -   c l a s s   A p p l i c a t i o n   - - - - -

class Application(Frame):
```

The constructor starts by calling its parent constructor and then calling a **`.__createWidgets()`** method to place all its widgets.

pageturnertest

```
# - - -   A p p l i c a t i o n . _ _ i n i t _ _   - - -

    def __init__ ( self, master=None ):
        Frame.__init__(self, master)
        self.grid()
        self.__createWidgets()
```

The **`.__createWidgets()`** method starts by instantiating a `PageTurner` widget and gridding it, then adding the usual Quit button.

pageturnertest

```
# - - -   A p p l i c a t i o n . _ _ c r e a t e W i d g e t s   - - -

    def __createWidgets(self):
        """Place all widgets in self."""
        self.pager  =  PageTurner ( self, size=(500,300) )
        self.pager.grid ( row=0, column=0 )
```

```
        self.quitButton  =  Button ( self, text="Quit",
            command=self.quit )
        self.quitButton.grid ( row=99, column=0, columnspan=99,
            sticky=E+W )
```

The content pages to be added to the `PageTurner` are very simple. Each consists of a single **`Label`** widget containing the string “This is page...”.

pageturnertest

```
        for i in range(1,9):
            content  =  Frame ( self.pager.bodyFrame )
            lab = Label ( content, text="This is page %d." % i )
            lab.grid(row=0, column=0, sticky=NW)
            self.pager.addPage ( content )
```

The main is the usual Tkinter main, instantiating the **`Application`** widget, setting its window name, and then entering the main loop.

pageturnertest

```
# - - - - -   m a i n   - - - - -

app  =  Application()
app.master.title ( "pageturnertest" )
app.mainloop()
```