npyscreen Documentation

Release 2

Nicholas Cole

Aug 04, 2017

Contents

Contents:

CHAPTER 1

An introduction to npyscreen

'Writing user interfaces without all that mucking about in hyperspace'

Purpose

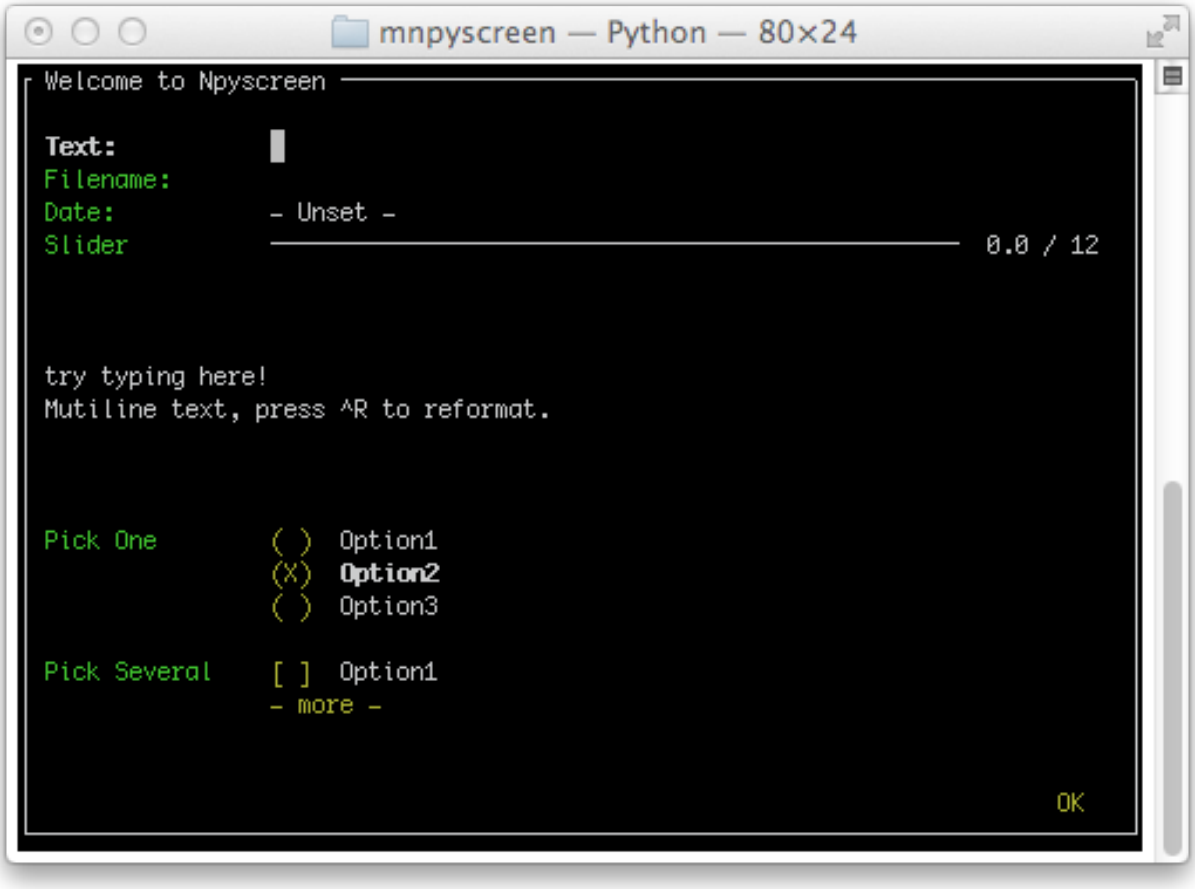
Npyscreen is a python widget library and application framework for programming terminal or console applications. It is built on top of ncurses, which is part of the standard library.

Wouldn't it be really nice if asking the user to give you some information could be easy? As easy as:

```
MyForm = Form()

usrn_box = MyForm.add_widget(TitleText, name="Your name:")
internet = MyForm.add_widget(TitleText, name="Your favourite internet page:")

MyForm.edit()

# usrn_box.value and internet.value now hold the user's answers.
```

If you think so, this library is for you.

Example Code



This is an example of a simple, one-screen application. More advanced applications would use the NPSAppManaged framework:

```
#!/usr/bin/env python
# encoding: utf-8

import npyscreen
class TestApp(npyscreen.NPSApp):
    def main(self):
        # These lines create the form and populate it with widgets.
        # A fairly complex screen in only 8 or so lines of code - a line for each␣
↪control.
        F  = npyscreen.Form(name = "Welcome to Npyscreen",)
        t  = F.add(npyscreen.TitleText, name = "Text:",)
        fn = F.add(npyscreen.TitleFilename, name = "Filename:")
        fn2 = F.add(npyscreen.TitleFilenameCombo, name="Filename2:")
        dt = F.add(npyscreen.TitleDateCombo, name = "Date:")
        s  = F.add(npyscreen.TitleSlider, out_of=12, name = "Slider")
        ml = F.add(npyscreen.MultiLineEdit,
               value = """try typing here!\nMutiline text, press ^R to reformat.\n""",
               max_height=5, rely=9)
        ms = F.add(npyscreen.TitleSelectOne, max_height=4, value = [1,], name="Pick␣
↪One",
```

```
                values = ["Option1","Option2","Option3"], scroll_exit=True)
        ms2= F.add(npyscreen.TitleMultiSelect, max_height =-2, value = [1,], name=
↪"Pick Several",
                values = ["Option1","Option2","Option3"], scroll_exit=True)

        # This lets the user interact with the Form.
        F.edit()

        print(ms.get_selected_objects())

if __name__ == "__main__":
    App = TestApp()
    App.run()
```

Strengths

This framework should be powerful enough to create everything from quick, simple programs to complex, multi-screen applications. It is designed to make doing the simple tasks very quick and to take much of the pain out of writing larger applications.

There is a very wide variety of default widgets - everything from simple text fields to more complex tree and grid views.

The focus of this library has always been to provide a rapid way to develop console applications. In general adding a control to the screen requires only one line of code.

Weaknesses

Version 2.0pre88 introduces the ability of Forms to resize themselves when the terminal size changes. Previous versions had always assumed a fixed-sized terminal.

Compatibility

Current development is done on Python 3, but the code is kept compatible with modern releases of Python 2. Certain features related to unicode work better under Python 3.

It is designed to run using only the python standard library, and the only requirements are a working python (2.6 or above) installation and a working curses library. Npyscreen will therefore work on almost all common platforms, and even in the Cygwin environment on Windows.

Python 3.4.0

There is a catastrophic bug in the Python curses module in 3.4.0: http://bugs.python.org/issue21088

This bug is fixed in Python 3.4.1, and I was not alerted to it until 3.4.1 was already out. I do not propose to release a workaround within npyscreen because I think the number of people who are stuck using Python 3.4.0 is very small. If this causes a problem for you, please get in touch.

Unicode

From version 2.0pre47 onwards all text widgets should now support utf-8 text display and entry on utf-8 capable terminals. This fixes a long-standing limitation with the library, and makes it suitable for use in projects targeting non-English-speaking users.

As of version 2.0pre48 the library aims to be robust in dealing with unicode across all widgets. There are still a few places in the system where support for utf-8/unicode needs further work. Please file bug reports if you encounter them.

Similar Projects

You might also like to look at http://excess.org/urwid/

Compared to npyscreen, urwid is more like a traditional, event-driven gui library, and targets other display devices as well as curses.

CHAPTER 2

Creating npyscreen applications

Objects Overview

Npyscreen applications are built out of three main types of object.

Form Objects Form objects (typically the size of a whole terminal, but sometimes larger or - for menus and the like - smaller) provide an area which can contain widget objects. They may provide additional functions like a system for handling menus, or routines that will be run if a user selects an "ok" button. They may also define operations carried out between key-presses, or as the user moves around the Form.

Widget Objects These are the individual controls on a form - text boxes, labels, sliders, and so on.

Application Objects These objects provide a convenient way to manage the running of your application. Although it is possible to program simple applications without using the Application objects, it is not advisable. Application objects make the management of multiple screens much less error-prone (in the 'bugs that may at any time crash your application') sense. In addition, making use of these objects will enable you to take advantage of additional features as npyscreen is developed.

Application structure for the impatient

Most new applications should look something like:

```
import npyscreen

# This application class serves as a wrapper for the initialization of curses
# and also manages the actual forms of the application

class MyTestApp(npyscreen.NPSAppManaged):
    def onStart(self):
        self.registerForm("MAIN", MainForm())

# This form class defines the display that will be presented to the user.
```

```
class MainForm(npyscreen.Form):
    def create(self):
        self.add(npyscreen.TitleText, name = "Text:", value= "Hellow World!" )

    def afterEditing(self):
        self.parentApp.setNextForm(None)

if __name__ == '__main__':
    TA = MyTestApp()
    TA.run()
```

Application structure in more detail (Tutorial)

First time users may find the code above confusing. If so, the following tutorial explains the structure of an npyscreen application in more detail. You should be able to follow it even if you know very little about the underlying curses system.

Forms, Widgets and Applications

Using a wrapper

Switiching into and out of a curses environment is a very boring task. The python curses module provides a wrapper to do this, and this is exposed by npyscreen as wrapper_basic. The basic framework for a very simple application looks like this:

```
import npyscreen

def myFunction(*args):
    pass

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print "Blink and you missed it!"
```

Which doesn't do anything clever. The curses environment starts and exits without actually doing anything. But it's a start.

Note that npyscreen also provides other wrappers that do slightly different things.

Using a Form

Now let's try putting something on the screen. For that we need a *Form* instance:

```
F = npyscreen.Form(name='My Test Application')
```

should do the trick. Let's put that into our wrapper:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
```

```
if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print "Blink and you missed it!"
```

Which still seems to do nothing – because we haven't actually displayed the Form. *F.display()* would put it on the screen, but we actually want to let the user play with it, so let's do F.edit() instead:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print "Blink and you missed it!"
```

Which won't run, because when we try to edit the Form npyscreen discovers there's no widget to edit. Let's put that right.

Adding the first widget

Let's put a textbox with a title in place. We do that with the code:

```
F.add(npyscreen.TitleText, name="First Widget")
```

The full code is:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    F.add(npyscreen.TitleText, name="First Widget")
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print "Blink and you missed it!"
```

Much better! That gives us something looking like an application. With just a three small changes we can change closing the message displayed to whatever the user typed:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    myFW = F.add(npyscreen.TitleText, name="First Widget")    # <------- Change 1
    F.edit()
    return myFW.value    # <------- Change 2

if __name__ == '__main__':
    print npyscreen.wrapper_basic(myFunction)   # <---- and change 3
```

Let's be a little more object-oriented

The approach we've been using works fine for simple applications, but once we start creating lots of widgets on a form, it is better to tuck all of that code away inside a Form object.

Instead of using the base Form() class in a very procedural way, let's create our own Form class. We'll override the Form's *create()* method, which is called whenever a Form is created:

```
class myEmployeeForm(npyscreen.Form):
    def create(self):
        super(myEmployeeForm, self).create()  # This line is not strictly necessary:␣
↪the API promises that the create method does nothing by default.
                                              # I've ommitted it from later example␣
↪code.
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment  = self.add(npyscreen.TitleText, name='Department')
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')
```

We can use our wrapper code from before to use it:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment  = self.add(npyscreen.TitleText, name='Department')
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

def myFunction(*args):
    F = myEmployeeForm(name = "New Employee")
    F.edit()
    return "Created record for " + F.myName.value

if __name__ == '__main__':
    print npyscreen.wrapper_basic(myFunction)
```

Offering Choice

Actually, we probably don't want just any old department name typed in - we want to offer a list of choices. Let's use the TitleSelectOne widget. It's a multi-line widget, so we need to take care that it takes up only a few lines of the screen (left to itself it would take up all the remaining space on the screen):

```
self.myDepartment = self.add(npyscreen.TitleSelectOne, max_height=3,
                             name='Department',
                             values = ['Department 1', 'Department 2', 'Department␣
↪3'],
                             scroll_exit = True  # Let the user move out of the␣
↪widget by pressing the down arrow instead of tab.  Try it without
                                                 # to see the difference.
                             )
```

Putting that in context:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def create(self):
```

```
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3
↪'])
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

def myFunction(*args):
    F = myEmployeeForm(name = "New Employee")
    F.edit()
    return "Created record for " + F.myName.value

if __name__ == '__main__':
    print npyscreen.wrapper_basic(myFunction)
```

Being Even More Object-Oriented

What we've done so far is all very well, but still ugly at the edges. We're still calling F.edit() ourselves, which is fine in a single-form application, but could lead to problems with recursion-depth later if we are not careful. It also prevents some of the more sophisticated features of the library from operating. The better solution is to use the *NPSAppManaged* class to manage your application.

Let's scrap the framework that has supported us so far, and start with a different basis for our application:

```
import npyscreen

class MyApplication(npyscreen.NPSAppManaged):
    pass

 if __name__ == '__main__':
    TestApp = MyApplication().run()
    print "All objects, baby."
```

Which will exit with an exception, because you have no 'MAIN' Form, which is the starting point for all NPSAppManaged applications.

Let's put that right. We'll use the Form class from before:

```
 import npyscreen

 class myEmployeeForm(npyscreen.Form):
     def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3
↪'])
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

class MyApplication(npyscreen.NPSAppManaged):
    def onStart(self):
        self.addForm('MAIN', myEmployeeForm, name='New Employee')

if __name__ == '__main__':
    TestApp = MyApplication().run()
    print "All objects, baby."
```

If you run the above code, you'll find yourself frustrated, because the application will continually display the form for you to edit, and you'll have to press "^C" (Control C) to exit.

That's because the NPSAppManaged class continually displays whatever form is named by its NEXT_ACTIVE_FORM attribute (in this case, the default - 'MAIN'). Older versions of this tutorial suggested setting that directly, but you should use the setNextForm(formid) method.

Let's alter the myEmployeeForm to tell it that after being run in an NPSAppManaged context, it should tell its NPSAppManaged parent to stop displaying Forms. We do that by creating the special method called *afterEditing*:

```
class myEmployeeForm(npyscreen.Form):
    def afterEditing(self):
        self.parentApp.setNextForm(None)

    def create(self):
       self.myName        = self.add(npyscreen.TitleText, name='Name')
       self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3
↪'])
       self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')
```

If we preferred, we could achieve the same result by defining a special method *onInMainLoop* in our MyApplication class - this method would get called after each form has been edited.

Our code now looks like this:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def afterEditing(self):
        self.parentApp.setNextForm(None)

    def create(self):
       self.myName        = self.add(npyscreen.TitleText, name='Name')
       self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3
↪'])
       self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

class MyApplication(npyscreen.NPSAppManaged):
   def onStart(self):
       self.addForm('MAIN', myEmployeeForm, name='New Employee')
       # A real application might define more forms here.......

if __name__ == '__main__':
   TestApp = MyApplication().run()
```

Choosing an approach

The last example above is probably over-kill for a very simple application. But it provides a much more robust framework with which to build larger applications than the framework we used at the start of the tutorial, at the cost of only a few lines of code. If you are displaying more than one screen, or running an application continuously, this is the approach you should take.

CHAPTER 3

Application Objects

NPSAppManaged provides a framework to start and end your application and to manage the display of the various Forms that you have created, in a way that should not create recursion depth problems.

Unless you have exceptionally good reasons to do otherwise, *NPSAppManaged* is almost certainly the best way to manage your application.

Unlike the plain NPSApp class, you do not need to write your own main loop - *NPSAppManaged* will manage the display of each Form of your application. Set up your form objects and simply call the *.run()* method of your NPSAppManaged instance.

Letting NPSAppManaged manage your Forms

There are three methods for registering a Form object with an NPSAppManaged instance:

`NPSAppManaged.`**`addForm`**`(`*id*, *FormClass*, ...`)`
: This version creates a new form and registers it with the NPSAppManaged instance. It returns a weakref.proxy to the form object. *id* should be a string that uniquely identifies the Form. *FormClass* should be the class of form to create. Any additional arguments will be passed to the Form's constructor. Use this version if you are not storing a separate reference to your form elsewhere.

`NPSAppManaged.`**`addFormClass`**`(`*id*, *FormClass* ...`)`
: This version registers a class of form rather than an instance. A new instance will be created every time it is edited. Additional arguements will be passed to the form's constructor every time it is created.

`NPSAppManaged.`**`registerForm`**`(`*id*, *fm*`)`
: *id* should be a string that uniquely identifies the form. *fm* should be a Form object. Note that this version only stores a weakref.proxy inside NPSAppManaged - in contrast to the .addForm version.

All Forms registered with an NPSAppManaged instance can access the controlling application as *self.parentApp*.

If for any reason you need to remove a Form, you can do with the *.removeForm(*id*)* method.

Running an NPSApplicationManaged application

run()
Start an NPSAppManaged application mainloop. This method will activate the default form, which should have been given an id of "MAIN".

NPSAppManaged.**STARTING_FORM**
If for any reason you need to change the name of the default form, you can change it here.

Once an application is running, the following methods control which form is presented to the user.

NPSAppManaged.**setNextForm**(*formid*)
Set the form to be displayed when the current one exits.

NPSAppManaged.**setNextFormPrevious**()
Set the form to be displayed when the current one exits to the previous one in the history

NPSAppManaged.**switchForm**(*formid*)
Immediately switch to the named form, bypassing any exit logic of the current form.

NPSAppManaged.**switchFormPrevious**()
Immediately switch to the previous form in the history.

In detail

Once all of your forms are ready and registered with an NPSAppManaged instance, you should call .run()

This method will activate the default form, which should have been given an id of "MAIN". You can change this default by changing the class/instance variable *.STARTING_FORM*.

Thereafter, the next form to be displayed will be the one specified by the instance variable *NEXT_ACTIVE_FORM*. Whenever a Form edit loop exits, the Form specified here will be activated. If *NEXT_ACTIVE_FORM* is None, the main loop will exit. *NEXT_ACTIVE_FORM* should be set by calling the application's *setNextForm(formid)* method. This documentation used to suggest that you set the attribute directly. While there are no immediate plans to deprecate this attribute, setting it directly should be avoided.

There are three mechanisms that Forms should use to control *NEXT_ACTIVE_FORM*.

1. All Forms registered with an NPSAppManaged which do *not* have the special method *.activate()* will have their method *.afterEditing* called, if they have it. Logic to determine which the *NEXT_ACTIVE_FORM* should be should go here. *NEXT_ACTIVE_FORM* should be set by calling the application's *setNextForm(formid)* method. If you are expecting your users to select an ok or cancel button, this is the preferred way to switch screens.
2. The application method *switchForm(formid)* causes the application to immediately stop editing the current form and switch to the one specified. Depending on the type of Form, the logic associated with them may be bypassed too.
3. Forms registered with an NPSAppManaged may be given an *.activate()* method, which NPSAppManaged will call instead of the usual *.edit()* method. This can contain additional logic. This is NOT the preferred method, but may allow greater flexibility. Note that in this case, the usual .edit() method will not be called, unless you call it explicitly. For example, an .activate() method might look like this:

```
def activate(self):
     self.edit()
     self.parentApp.setNextForm(None)
```

which would cause the mainloop to exit after the Form was complete.

Additional Services offered by NPSAppManaged

The following methods may be usefully overridden by subclassing NPSAppManaged. By default they do nothing.

NPSAppManaged.**onInMainLoop**()
: Called between each screen while the application is running. Not called before the first screen.

NPSAppManaged.**onStart**()
: Override this method to perform any initialisation. If you wish, you can set up your application's Forms here.

NPSAppManaged.**onCleanExit**()
: Override this method to perform any cleanup when application is exiting without error.

NPSAppManaged.**keypress_timeout_default**
: If this is set, new forms will be created with keypress_timeout set to this, provided they know what application they belong to - i.e. they have been passed *parentApp*= at creation time. If you are using NPSAppManaged, this will happen automatically.

NPSAppManaged.**while_waiting**()
: Applications can also have a *while_waiting* method. You can define and override this at will, and it will be called while the application is waiting for user input (see the while_waiting method on forms).

NPSAppManaged.**_internal_while_waiting**()
: This method is for internal use by npyscreen.

NPSAppManaged.**switchForm**(*formid*)
: Immediately stop editing the current form and switch to the specified form.

NPSAppManaged.**switchFormPrevious**()
: Immediately switch to the previous form in the history.

NPSAppManaged.**resetHistory**()
: Forget the previous forms visted.

NPSAppManaged.**getHistory**()
: Get a list of the Forms visited

Methods and attributes on Forms managed by this class

Forms called by NPSAppManaged can be given the methods

Form.**beforeEditing**()
: called before the edit loop of the form is called

Form.**afterEditing**()
: called when the form is exited

Form.**activate**()
: The presence of this method entirely overrides the existing .beforeEditing .edit and afterEditing methods.

CHAPTER 4

Other Application classes

class `NPSApp`

To use NPSApp subclass it and provide your own *.main()* definition. When you are ready to run the application call *.run()* and your mainloop will be executed.

While it provides maximum flexibility, NPSApp is in almost every other way inferior to NPSAppManaged. Do not use it for new projects, and reguard it as an internal base class only.

CHAPTER 5

Form Objects

A Form object is a screen area that contains widgets. Forms control which widget a user is editing, and may provide additional functionality, such as pop-up menus or actions that happen on particular keypresses.

Creating a Form

class `Form` (*name=None*, *lines=0*, *columns=0*, *minimum_lines=24*, *minimum_columns=80*)

Forms have the following class attributes:

```
DEFAULT_LINES      = 0
DEFAULT_COLUMNS    = 0
SHOW_ATX           = 0
SHOW_ATY           = 0
```

The default values will create a form that fills the whole screen and which is displayed in the top left corner. See the arguments passed in to the constructor for more details on controlling the size of a form.

The Following arguments can be passed to a Form's constructor:

name= Names the Form. As for some widgets, this will display a title.

lines=0, columns=0, minimum_lines=24, minimum_columns=80 You can adjust the size of the Form, either providing an absolute size (with *lines=* and *columns=*) or a minimum size (*minimum_lines=* and *minimum_columns=*). The default minimums (24x80) provide the standard size for terminal. If you plan your Forms to fit within that size, they should be viewable on almost all systems without the need to scroll the Form. Note that you can use the absolute sizing in one direction and the minimum in the other, should you wish.

The standard constructor will call the method *.create()*, which you should override to create the Form widgets. See below.

Placing widgets on a Form

To add a widget to a Form, use the method:

`Form.`**`add`**`(`*WidgetClass*, ...`)`
WidgetClass must be a class, all of the additional arguments will be passed to the widget's own constructor. A reference to the widget will be returned.

The position and size of a widget are controlled by the widget's constructor. However, there are hints that the Form class provides. If you do not override the position of the widget, it will be placed according to the Form's *.nextrelx* and *nextrely* instance attributes. The *.nextrely* attribute is increased automatically each time a widget is placed. You might also increase it yourself by doing something like:

```
self.nextrely += 1
```

Which would leave a gap between the previous widget and the next placed one.

`Form.`**`nextrely`**
The y position at which the next created widget should be made. The standard forms set this to the line below the previously created widget as each widget is added to the form.

`nextrelx`
The x position at which the next created widget should be made.

Other Standard Form Features

`Form.`**`create`**`()`
This method is called by the Form's constructor. It does nothing by default - it is there for you to override in subclasses, but it is the best place to set up all the widgets on a Form. Expect this method to be full of *self.add(...)* method calls, then!

`Form.`**`while_editing`**`()`
This method is called as the user moves between widgets. It is intended for you to override in subclasses, to do things like altering one widget based on the value of another.

`Form.`**`adjust_widgets`**`()`
Be very careful with this method. It is called for every keypress while the Form is being edited, and there is no guarantee that it might not be called even more frequently. By default it does nothing, and is intended for you to override. Since it gets called so frequently, thoughtlessness here could slow down your whole application.

For example, be very conservative with redraws of the whole Form (a slow operation) - make sure you put in code to test whether a redraw is necessary, and try to only redraw widgets that really need to be changed, rather than redrawing the whole screen.

If the Form's parentApp also has a method called *adjust_widgets*, this will also be called.

`Form.`**`while_waiting`**`()`
If you wish to perform actions while waiting for the user to press a key, you may define a *while_waiting* method. You should also set the attribute *keypress_timeout*, which is a value in ms. Whenever waiting for input, if more than the time given in *keypress_timeout* passes, while_waiting will be called. Note that npyscreen takes no steps to ensure that *while_waiting()* is called at exactly regular intervals, and in fact it may never be called at all if the user continually presses keys.

If a form's parentApp has a method called *while_waiting* this will also be called.

A *keypress_timeout* value of 10 suggests that the *while_waiting* method is called about every second, assuming the user takes no other action.

See the included example Example-waiting.py for a fully worked example.

Form.**keypress_timeout**
See the *while_waiting* method above.

Form.**set_value**(*value*)
Store *value* in the *.value* attribute of the *Form* and then call the *whenParentChangeValue* method of every widget that has it.

Displaying and Editing Forms

Form.**display**()
Redraw every widget on the Form and the Form itself.

Form.**DISPLAY**()
Redraw the form, but make extra sure that the display is reset. This is a slow operation, and avoid calling if possible. You may sometimes need to use this if an external process has disrupted the terminal.

Form.**edit**()
Allow the user to interactively edit the value of each widget. You should not need to call this method if correctly using the *NPSAppManaged* class. You should avoid calling this method if possible, but you will need to use it if writing simple applications that do not use the NPSAppManaged class. Calling this method directly is akin to creating a modal dialog box in a GUI application. As far as possible consider this method an internal API call.

When forms exit

Forms may exit their editing modes for a number of reasons. In NPSAppManaged applications, the controlling application may cause the form to exit.

Setting the attribute *.editing* to False yourself, however, will cause the form to exit.

Standard Form Classes

class Form
The basic Form class. When editing the form, the user can exit by selecting the OK button in the bottom right corner.

By default, a Form will fill the Terminal. Popup is simply a Form with a smaller default size.

class Popup
Popup is simply a Form with a smaller default size.

class ActionForm
The ActionForm creates OK and Cancel buttons. Selecting either exits the form. The method *on_ok* or *on_cancel* is called when the Form exits (assuming the user selected one of these buttons). Subclasses may therefore usefully override one or both of these methods, which by default do nothing.

on_ok()
Called when the ok button is pressed. Setting the attribute *.editing* to True in this method will abort editing the form.

on_cancel()
Called when the cancel button is pressed. Setting the attribute *.editing* to True in this method will abort editing the form.

class ActionFormV2
New in Version 4.3.0. This version of ActionForm behaves similarly to ActionForm above, but the code is much cleaner. It should be much easier to subclass. Eventually, this version may entirely replace ActionForm.

class ActionFormMinimal
New in Version 4.4.0. This version of ActionFormV2 only features an OK button. Added at user request for use in special circumstances.

class ActionPopup
A smaller version of the ActionForm.

class SplitForm
The SplitForm has a horizontal line across the middle. The method *get_half_way()* will tell you where it has been drawn.

draw_line_at
This attribute defines the position at which the line should be drawn across the screen. It can be set by passing *draw_line_at*= to the constructor, or will be set automatically at the value returned by the method *get_half_way*.

get_half_way()
return the y co-ordinate of the bar across the middle of the form. In fact in subclasses of this form, there is no particular reason why the y co-ordinate should in fact be half way down the form, and subclasses may return whatever value is convenient.

MOVE_LINE_ON_RESIZE
This class attribute specifies whether the position of the line should be moved when the form is resized. Since any widgets below the line would also need to be moved (presumably in an overriden *resize* method on subclasses of this form, this value is set to False by default).

class FormWithMenus
Similar to the Form class, but provides the additional functionality of Popup menus.

To add a new menu to the Form use the method *new_menu(name='')*. This will create the menu and return a proxy to it. For more details see the section on Menus below.

class ActionFormWithMenus
Similar to the ActionForm class, but provides the additional functionality of Popup menus.

To add a new menu to the Form use the method *new_menu(name='')*. This will create the menu and return a proxy to it. For more details see the section on Menus below.

class ActionFormV2WithMenus
New in Version 4.3.0. This version of ActionFormWithMenus behaves similarly to ActionForm above, but the code is much cleaner. It should be much easier to subclass. Eventually, this version may entirely replace ActionFormWithMenus.

class FormBaseNew
This form does not have an *ok* or *cancel* button by default. The additional methods *pre_edit_loop* and *post_edit_loop* are called before and after the Form is edited. The default versions do nothing. This class is intended as a base for more complex user interfaces.

pre_edit_loop()
Called before the form is edited.

post_edit_loop()
Called after the edit loop exits.

class FormBaseNewWithMenus
Menu-enabled version of FormBaseNew.

Mutt-like Forms

class `FormMutt`

Inspired by the user interfaces of programs like *mutt* or *irssi*, this form defines four default widgets:

wStatus1 This is at the top of the screen. You can change the type of widget used by changing the *STATUS_WIDGET_CLASS* class attribute (note this is used for both status lines).

wStatus2 This occupies the second to last line of the screen. You can change the type of widget used by changing the *STATUS_WIDGET_CLASS* class attribute (note this is used for both status lines).

wMain This occupies the area between wStatus1 and wStatus2, and is a MultiLine widget. You can alter the type of widget that appears here by subclassing *FormMutt* and changing the *MAIN_WIDGET_CLASS* class attribute.

wCommand This Field occupies the last line of the screen. You can change the type of widget used by altering the *COMMAND_WIDGET_CLASS* class attribute.

By default, wStatus1 and wStatus2 have *editable* set to False.

FormMuttActive, FormMuttActiveWithMenus, FormMuttActiveTraditional, FormMuttActiveTraditionalWithMenus

These classes are intended to make the creation of more complicated applications easier. Each class uses the additional classes *NPSFilteredDataBase*, *ActionControllerSimple*, *TextCommandBox*, *TextCommandBoxTraditional*.

A very common *nix style of terminal application (used by applications like mutt and irssi) has a central display with a list or grid of times, a command line at the bottom and some status lines.

These classes make setting up a similar form easy. The difference between the *FormMuttActive* and *FormMuttActiveTraditional* classes is that in the latter the only widget that the user ever actually edits is the command line at the bottom of the screen. However, keypresses will be passed to the multiline widget in the centre of the display if these widgets are not editing a command line, allowing the user to scroll around and select items.

What is actually displayed on the screen is controlled by the *ActionControllerSimple* class, which uses as a base the data stored not by any of the individual widgets but by the *NPSFilteredDatabase* class.

See the section on writing Mutt-like applications later in this documentation for more information.

Multi-page Forms

class `FormMultiPage` (*new in version 2.0pre63*)

This *experimental* class adds support for multi-page forms. By default, scrolling down off the last widget on a page moves to the next page, and moving up from the first widget moves back a page.

The default class will display the page you are on in the bottom right corner if the attribute *display_pages* is True and if there is more than one page. You can also pass *display_pages=False* in to the constructor. The color used for this display is stored in the attribute *pages_label_color*. By default this is 'NORMAL'. Other good values might be 'STANDOUT', 'CONTROL' or 'LABEL'. Again, you can pass this in to the constructor.

Please note that this class is EXPERIMENTAL. The API is still under review, and may change in future releases. It is intended for applications which may have to create forms dynamically, which might need to create a single form larger than a screen (for example, a Jabber client that needs to display an xmpp form specified by the server.) It is *not* intended to display arbitrarily large lists of items. For that purpose, the multiline classes of widgets are much more efficient.

Three new methods are added to this form:

FormMultiPage.**add_page**()
Intended for use during the creation of the form. This adds a new page, and resets the position at which new widgets will be added. The index of the page added is returned.

FormMultiPage.**switch_page**(**index**)
This method changes the active page to the one specified by *index*.

FormMultiPage.**add_widget_intelligent**(**args*, ***keywords*)
This method adds a widget to the form. If there is not enough space on the current page, it tries creating a new page and adding the widget there. Note that this method may still raise an exception if the user has specified options that prevent the widget from appearing even on the new page.

class FormMultPageAction(*new in version 2.0pre64*)
This is an *experimental* version of the FormMultiPage class that adds the on_ok and on_cancel methods of the ActionForm class and automatically creates cancel and ok buttons on the last page of the form.

class FormMultiPageWithMenus
Menu-enabled version of MultiPage.

class FormMultiPageActionWithMenus
Menu-enabled version of MultiPageAction.

Menus

Some Form classes support the use of popup menus. Menus could in theory be used as widgets on their own. Popup menus (inspired, in fact, by the menu system in RiscOS) were selected instead of drop-down menus as being more suitable for a keyboard environment, making better use of available screen space and being easier to deploy on terminals of varied sizes.

By default, the supporting forms will display an advert that the menu system is available to the user, and a shortcut to the list of menus. If the form has multiple menus, a 'root' menu listing all of them will be displayed.

Menus are usually created by calling a (supporting) Form's *new_menu* method. Version 2.0pre82 adds the argument *shortcut=None* to this method. In the list of menus that the Form displays, this shortcut will be displayed. After a menu has been created, the following methods on that object are useful:

NewMenu.**addItem**(*text=''*, *onSelect=function*, *shortcut=None*, *arguments=None*, *keywords=None*)
text should be the string to be displayed on the menu. *onSelect* should be a function to be called if that item is selected by the user. This is one of the few easy opportunities in npyscreen to create circular references - you may wish to pass in a proxy to a function instead. I've tried to guard you against circular references as much as possible - but this is just one of those times I can't second-guess your application structure. Version 2.0pre82 adds the ability to add a shortcut.

From version 3.6 onwards, menu items can be specified with a list of *arguments* and/or a dictionary of keywords.

NewMenu.**addItemsFromList**(*item_list*)
The agument for this function should be a list or tuple. Each element of this should be a tuple of the arguments that are used for creating each item. This method is DEPRECATED and may be removed or altered in a future version.

NewMenu.**addNewSubmenu**(*name=None*, *shortcut=None*, *preDisplayFunction=None*, *pdfuncArguments=None*, *pdfuncKeywords=None*)
Create a new submenu (returning a proxy to it). This is the preferred way of creating submenus. Version 2.0pre82 adds the ability to add a keyboard shortcut.

From version 3.7 onwards, you can define a function and arguments to be called before this menu is displayed. This might mean you can adjust the content of the menu at the point it is displayed. Added at user request.

NewMenu.**addSubmenu**(*submenu*)

Add an existing Menu to the Menu as a submenu. All things considered, addNewSubmenu is usually a better bet.

(Internally, this menu system is referred to as the "New" menu system - it replaces a drop-down menu system with which I was never very happy.)

Resizing Forms (New in version 2.0pre88)

When a form is resized, a signal is sent to the form currently on the screen. Whether or not the form handles this is decided by three things.

If you set the variable *npyscreen.DISABLE_RESIZE_SYSTEM* to True, forms will not resize at all.

The class attribute *ALLOW_RESIZE* (=True by default). If this is set to false the form will not resize itself.

The class attribute *FIX_MINIMUM_SIZE_WHEN_CREATED* controls whether the form can be made smaller than the size it was when it was created. By default this is set to *False*. This is because for over a decade, npyscreen assumed that forms would never change size, and many programs may rely on the fact that the form will never be resized. If you are writing new code from scratch, you can set this value to True, provided that you test the results to make sure that resizing the form will not crash your application.

When a form is resized, the method *resize* will be called *after* the new size of the form has been fixed. Forms may override this method to move widgets to new locations or alter anything else about the layout of the form as appropriate.

When using the *NPSAppManaged* system, forms will be automatically resized before they are displayed.

CHAPTER 6

Widgets: Basic Features

Creating Widgets

Widgets are created by passing their class as the first argument of a Form's *add(...)* method. The remaining arguments will be passed to the widget's own constructor. These control things such as size, position, name, and initial values.

Constructor arguments

name= You should probably give each widget a name (a string). Where appropriate, it will be used as the label of the widget.

relx=, rely= The position of the widget on the Form is controlled by relx and rely integers. You don't have to specify them, in which case the form will do its best to decide where to put the widget. You can specify only one or the other if you so choose (eg. you probably don't usually need to specify relx). *New in Version 4.3.0*: if you give a negative value for rely or relx, the widget will be positioned relative to the bottom or right hand side of the Form. If the form is resized, npyscreen will do its best to keep the widget in place.

width=, height=, max_width=, max_height= By default, widgets will expand to fill all available space to the right and downwards, unless that would not make sense - for example single lines of text do not need moe than one line, and so don't claim more than one. To alter the size of a widget, therefore, specify a different *max_width* or *max_height*. It is probably better to use the max_ versions - these will not raise an error if space is getting tight and you specify too much, but will try to squash the widget into remaining space.

value= The value of a widget is the thing a user can change - a string, a date, a selection of items, a filename. The initial setting of the *.value* attribute can be specified here.

values= Where a widget offers the user a selection from a list of values, these can be specified here: this is the initial setting of the *values* attribute.

editable=True Whether the user should be able to edit a widget. (Initial setting of the *.editable* attribute.)

hidden=False Whether a widget is visible or not. (Initial setting of the *.hidden* attribute.)

color='DEFAULT', labelColor='LABEL' Provides a hint to the colour-management system as to how the widget should be displayed.

For more details see *setting up colors*.

scroll_exit=False, slow_scroll=False, exit_left, exit_right These affect the way a user interacts with multi-line widgets. *scroll_exit* decides whether or not the user can move from the first or last item to the previous or next widget. *slow_scroll* means that widgets that scroll will do so one line at at time, not by the screen-full. The options *exit_left|right* dictate whether the user can exit a widget using the left and right arrow keys.

Using and Displaying Widgets

All widgets have the following methods:

display() Redraws the widget and tells curses to update the screen.

update(clear=True) Redraws the widget, but doesn't tell curses to update the screen (it is more efficient to update all widgets and then have the Form on which they sit tell curses to redraw the screen all in one go).

Most widgets accept the optional argument *clear=False|True* which affects whether they first blank the area they occupy before redrawing themselves.

when_parent_changes_value() Called whenever the parent form's *set_value(value)* method is called.

when_value_edited() Called when, during editing of the widget, its value changes. I.e. after keypresses. You can disable this by setting the attribute *check_value_change* to False.

You can override this function for your own use.

when_cursor_moved() Called when, during the editing of the widget, its cursor has been moved. You can disable the check for this by setting the attribute *check_cursor_move* to False.

You can override this function for your own use.

edit() Allow the user to interact with the widget. The method returns when the user leaves the widget. In most cases, you will never need to call this method yourself, and for the most part this should be regarded as part of the internal API of npyscreen.

set_relyx() Set the position of the widget on the Form. If y or x is a negative value, npyscreen will try to position it relative to the bottom or right edge of the Form. Note that this ignores any margins that the Form may have defined. (New in Version 4.3.0).

safe_to_exit() This method is called by the default handlers before a user tries to exit from a widget. It should return True if this should be allowed and False if it should not. You may override this method to perform any verification of the contents of a field before allowing the user to exit. (New in Version 4.10.0)

Titled Widgets

Many widgets exist in two forms, one with a label, one without. For example Textbox and TitleText. If the label is particularly long (at time of construction), the label may be put on its own line. Additional constructor arguments:

use_two_lines= If either True or False, override what the widget would otherwise choose.

field_width= (For text fields) - how wide should the entry part of the widget be?

begin_entry_at=16 At what column should the entry part of the widget begin?

Internally titled widgets are actually a textbox (for the label) and whatever other kind of widget is required. You can access the separate widgets (if you ever need to - you shouldn't) through the *label_widget* and *entry_widget* attributes. However, you may never need to, since the *value* and *values* attributes of the combined widget should work as expected.

Creating your own widgets

All widgets should inherit from the class *Widget*.

calculate_area_neeeded This function is called to ask the widget how many lines and columns it requires (for a minimal display). You should return a tuple with exactly two numbers. Returning 0 for either argument says that the widget should be given all the remaining space on the display if it is available.

If you are writing text to the screen you should avoid using curses directly, and instead use the function

add_line(realy, realx, unicode_string, attributes_list, max_columns, force_ascii=False) This function adds a line of text to the display. *realy* and *realx* are the absolute position on the Form. *attributes_list* is a list of attributes that should be applied to each character. If all of them require the same attribute, use the *make_attributes_list* method to create a list of the right length.

make_attributes_list(unicode_string, attribute) A convenience function. Retuns a list the length of the unicode_string provided, with each entry of the list containing a copy of attribute.

resize() You can override this method to perform any necessary actions when the widget is resized. (New in version 4.3.0)

CHAPTER 7

Widgets: Displaying Text

Textfild, TitleText A single line of text, although of arbitrary length - the basic entry widget. N.B. some versions of the documentation incorrection refer to a 'textbox' widget.

FixedText, TitleFixedText A single line of text, but with the editing functions of Textbox removed.

PasswordEntry, TitlePassword A textbox but altered so that the exact letters of *.value* are not displayed.

Autocomplete This is a textbox but with additional functionality - the idea is that if the user presses TAB the widget will attempt to 'complete' what the user was typing, offering a choice of options if appropriate. The method called is *auto_complete(inputch)*.

Of course, context is everything here. *Autocomplete* is therefore not useful, but is intended as something you can subclass. See the Filename and TitleFilename classes for examples.

Filename, TitleFilename A textbox that will attempt to 'complete' a filename or path entered by the user.

This is an example of the *Autocomplete* widget.

FilenameCombo, TitleFilenameCombo This is a more advanced way to select files. New in version 2.0pre82.

MultiLineEdit This widget allows the user to edit several lines of text.

Pager, TitlePager This widget displays lines of text, allowing the user to scroll through them, but not edit them. The text to display is held in the *.values* attribute.

In detail

class `Textbox`

`display_value`(*vl*)
Control how the value of the *.value* attribute is displayed. Since versions of the text widgets are used in other, compound widgets (such as most of the multiline classes), this method is often overridden.

show_brief_message()
Beep and display a brief message to the user. In general, there are better ways to do this, but this is somtimes useful, for example when showing errors in Autocomplete classes.

CHAPTER 8

Widgets: Picking Options

MultiLine Offer the user a list of options. (This widget could probably have a better name, but we're stuck with it for now)

The options should be stored in the attribute *values* as a list. The attribute *value* stores the index of the user's selection. If you want to return the actual selected values rather than an index, use the *get_selected_objects()* method.

One of the most important features of MultiLine and widgets derived from it is that it can be adapted easily to allow the user to choose different types of objects. To do so, override the method *display_value(self, vl)*. The argument *vl* will be the object being displayed, and the function should return a string that can be displayed on the screen.

In other words you can pass in a list of objects of arbitrary types. By default, they will be displayed using *str()*, but by overriding *display_value* you can present them however you see fit.

MultiLine also allows the user to 'filter' entries. (bound to keys l, L, n, p by default for filter, clear filter, next and previous). The current implementation highlights lines that match on the screen. Future implementations may hide the other lines or offer a choice. You can control how the filter operates by overriding the filter_value method. This should accept an index as an argument (which looks up a line in the list .values) and should return True on a match and False otherwise. From version 2.0pre74, the whole filtering system can be disabled by setting that attribute *.allow_filtering* to False. This can also be passed in as an argument to the constructor.

> Multiline widgets are a container widget that then holds a series of other widgets that handle various parts of the display. All multiline classes have a *_contained_widget* class attribute. This controls how the widget is constructed. The class attribute *_contained_widget_height* specifies how many lines of the screen each widget should be given.

TitleMultiLine A titled version of the MultiLine widget.

If creating your own subclasses of MultiLine, you can create Title versions by subclassing this object and changing the *_entry_type* class variable.

MultiSelect, TitleMultiSelect, Offer the User a list of options, allow him or her to select more than one of them.

The *value* attribute is a list of the indexes user's choices. As with the MultiLine widget, the list of choices is stored in the attribue *values*.

SelectOne, TitleSelectOne Functionally, these are like the Multiline versions, but with a display similar to the MultiSelect widget.

MultiSelectFixed, TitleMultiSelectFixed These special versions of MultiSelect are intended to display data, but like Textfixed do not allow the user to actually edit it.

MultiLineAction A common use case for this sort of widget is to perform an action on the currently highlighted item when the user pushes Return, Space etc. Override the method *actionHighlighted(self, act_on_this, key_press)* of this class to provide this sort of widget. That method will be called when the user 'selects' an item (though in this case .value will not actually be set) and will be passed the item highlighted and the key the user actually pressed.

MultiSelectAction This is similar to the MultiLineAction widget above, except that it also provides the method *actionSelected(self, act_on_these, keypress)*. This can be overridden, and will be called if the user pressed ';'. The method will be passed a list of the objects selected and the keypress. You probably want to adjust the default keybindings to make this widget useful.

BufferPager, TitleBufferPager ***New in Version 2.0pre90*** The *BufferPager* class is a subclass of the *Pager* class. It is designed to display text to the user in much the way that *tail -f* does under *nix. By default, the .values attribute is set to an instance of the *collections.deque* class. You can pass a *maxlen=* value to the constructor. If not, the maxlen for the deque object will be taken from the class attribute *DEFAULT_MAXLEN*, which is None by default.

`BufferPager.`**`clearBuffer`**`()`
Clear the buffer.

`BufferPager.`**`buffer`**`(`*lines*, *scroll_end=True*, *scroll_if_editing=False*`)`
Add *lines* to the contained deque object. If *scroll_end* is True, scroll to the end of the buffer. If *scroll_if_editing* is True, then scroll to the end even if the user is currently editing the Pager. If the contained deque object was created with a maximum length, then new data may cause older data to be forgotten.

MultiLineEditable A list of items that the user can edit, based on the multiline classes. New in version 3.9

`get_new_value``()`
This method should return a 'blank' object that can be used to initialize a new item on the list. By default it returns an empty string.

MultiLineEditableTitle A titled version of MultiLineEditable. The class attribute *_entry_type* controls the type of contained widget.

MultiLineEditableBoxed A boxed version of MultiLineEditable. The class attribute **_entry_type* controls the type of contained widget.

Custom Multiselect Widgets

Multiline widgets are a container widget that then holds a series of other widgets that handle various parts of the display. All multiline classes have a *_contained_widget* class attribute. This controls how the widget is constructed. The class attribute *_contained_widget_height* specifies how many lines of the screen each widget should be given.

From version 3.4 onwards, contained widgets that have a *.selected* attribute are handled differently: widgets will have their *.selected* attribute set to *True* if the line is selected and *False* otherwise. Widgets may also have their *.important* attribute set to True or False, depending on if they are included in a current filter (see above).

Widgets that do not have a *selected* attribute have the value for each line put in their *name* attribute, and whether the line is selected or not put in their *value* attribute. This is a legacy of the fact that the standard multiselect widgets use checkboxes to display each line.

From version 4.8.7 onwards, multiline widgets use the methods *set_is_line_important*, *set_is_line_bold* and *set_is_line_cursor* to control the display of each line. These methods are passed the widget object in question and a Boolean value. They are intended to be overridden.

CHAPTER 9

Widgets: Trees and Tree displays

Representing Tree Data

TreeData The TreeData class is used to represent tree objects. Each node of the tree, including the root node, is an NPSTreeData instance. Each node may have its own content, a parent or children.

The content of each node is either set when it is created or using the *.set_content* method.

get_content() returns the content.

get_content_for_display() is used by the widgets that display trees, which expect it to return a string that can be displayed to the user to represent the content. You might want to overload this method.

new_child(content=...) creates a new child node.

selectable If this attribute is true the user can mark a value as 'selected'. This is used by MLTreeMultiSelect widget, and is True by default.

ignore_root This attribute controls whether the root node of the tree is displayed to the user.

expanded This attribute controls whether this branch of the tree is expanded, assuming the node has any children.

sort This attribute controls whether the tree should be sorted.

sort_function If the tree is sorted, the function named in this attribute will be used as a key to sort the tree when it is being displayed.

walk_tree(only_expanded=True, ignore_root=True, sort=None, sort_function=None) Iterate over the tree. You can override the standard sort and sort functions, and decide whether or not to iterate over only nodes of the tree that are marked as expanded.

Trees

MLTree, MLTreeAction The *values* attribute of this class must store an NPSTree instance. However, if you wish you can override the method *convertToTree* of this class. This method should return an NPSTree instance. The function will be called automatically whenever *values* is assigned.

By default this class uses *TreeLine* widgets to display each line of the tree. In derived classes You can change this by changing the class attribute *_contained_widgets*. The class attribute *_contained_widget_height* specifies how many lines of the screen each widget should be given.

MLTreeAnnotated, MLTreeAnnotatedAction By default this class uses *TreeLineAnnotated* widgets to display each line of the tree. In derived classes You can change this by changing the class attribute *_contained_widgets*.

MLTreeMultiSelect *New in version 2.0pre70*

This class allows you to select multiple items of a tree. You can select which nodes of NPSTreeData the user is able to select by setting the attribute *selectable* on that node.

The method *get_selected_objects(self, return_node=True,)* returns an generator object that lists the nodes that are selected. If return_node is True, the actual node itself is yielded, otherwise the value of *node.getContent()* is yielded instead.

New in Version 2.0pre71 If the attribute *select_cascades* is True (which can be set by passing the argument *select_cascades* at the time of creation or setting the attribute directly later), selecting a node will automatically select any selectable nodes under the selected node. This is set to True by default.

The selected nodes also have their attribute *selected* set to True, and so you can walk through the tree to find them if you prefer.

The widget used to display each line is *TreeLineSelectable*.

MLTreeMultiSelectAnnotated *New in version 2.0pre71*

A version of the MLTreeMultiSelect class that uses *TreeLineSelectableAnnotated* as its display widgets.

Deprecated Tree Classes

NPSTreeData DEPRECATED in favour of the TreeData class. The NPSTreeData class is used to represent tree objects. Each node of the tree, including the root node, is an NPSTreeData instance. Each node may have its own content, a parent or children.

The content of each node is either set when it is created or using the *.setContent* method.

.getContent returns the content.

.getContentForDisplay is used by the widgets that display trees, which expect it to return a string that can be displayed to the user to represent the content. You might want to overload this method.

newChild(content=...) creates a new child node.

selectable (new in version 2.0pre70) If this attribute is true the user can mark a value as 'selected'. This is used by MLTreeMultiSelect widget, and is True by default.

MultiLineTree, SelectOneTree, and MultiLineTree These widgets all work in a very similar way to the non-Tree versions, except that they expect to contain an NPSTree in their .values attribute. The other major difference is that their .value attribute does not contain the index of the selected value(s), but the selected value(s) itself/themselves. However, these classes are DEPRECATED in favour of the much improved *MLTree* and *MLTreeAction* classes.

MultiLineTreeNew, MultiLineTreeNewAction *These classes are provided solely for compatibility with old code. New classes should use the MLTree and related classes*

The *values* attribute of this class must store an NPSTree instance. However, if you wish you can override the method *convertToTree* of this class. This method should return an NPSTree instance. The function will be called automatically whenever *values* is assigned.

By default this class uses *TreeLineAnnotated* widgets to display each line of the tree. In derived classes You can change this by changing the class attribute *_contained_widgets*.

MutlilineTreeNewAnnotated, MultilineTreeNewAnnotatedAction *These classes are provided solely for compatibility with old code. New classes should use the MLTree and related classes*

By default this class uses *TreeLineAnnotated* widgets to display each line of the tree. In derived classes You can change this by changing the class attribute *_contained_widgets*.

CHAPTER 10

Widgets: Dates, Sliders and Combination Widgets

DateCombo, TitleDateCombo These widgets allow a user to select a date. The actual selection of a date is done with the class MonthBox, which is displayed in a temporary window. The constructor can be passed the following arguments - *allowPastDate=False* and *allowTodaysDate=False* - both of which will affect what the user is allowed to select. The constructor can also accept the argument *allowClear=True*.

ComboBox, TitleCombo This box looks like a Textbox, but the user can only select from a list of options. Which are displayed in a temporary window if the user wants to change the value. Like the MultiLine widget, the attribute *value* is the index of a selection in the list *values*. The ComboBox widget can also be customised by overloading the *display_value(self, vl)* method.

FilenameCombo, TitleFilenameCombo This presents a control for picking filenames. The following arguments can be passed in to the constructor:

```
select_dir=False
must_exist=False
confirm_if_exists=False
sort_by_extension=True
```

These also correspond to attributes on the widget that can be set at any time.

This widget was added in version 2.0pre81.

Slider, TitleSlider Slider presents a horizontal slider. The following additional arguments to the constructor are useful:

out_of=100 The maximum value of the slider.

step=1 The increments by which a user my increase or decrease the value.

lowest=0 The minimum value a user can select. Note that sliders are not designed to allow a user to select negative values. *lowest* should be >= 0

label=True Whether to print a text label next to the slider. If so, see the *translate_value* method.

block_color = None The colour of the blocks that show the level of the slider. By default (None) the same value as the colour of the slider itself.

All of these options set attributes of the same name that may be altered once the widget exists.

The text displayed next to the widget (if *label=True*) is generated by the *translate_value* method. This takes no options and returns a string. It makes sense to subclass the Slider object and overload this method. It probably makes sense to ensure that the string generated is of a fixed length. Thus the default code looks like:

```
stri = "%s / %s" %(self.value, self.out_of)
l = (len(str(self.out_of)))*2+4
stri = stri.rjust(l)
return stri
```

SliderNoLabel, TitleSliderNoLabel These versions of the Slider do not display a label. (Similar to using the usual slider with *label=False*). New in Version 4.3.5

SliderPercent, TitleSliderPercent These versions of the Slider display a percentage in the label. The number of decimal places can be set by setting the attribute *accuracy* or by passing in the keyword *accuracy* to the constructor. Default is 2. New in Version 4.3.5.

CHAPTER 11

Widgets: Grids

SimpleGrid This offers a spreadsheet-like display. The default is only intended to display information (in a grid of text-fields). However, it is designed to be flexible and easy to customize to display a variety of different data. Future versions may include new types of grids. Note that you can control the look of the grid by specifying either *columns* or *column_width* at the time the widget is created. It may be that in the future the other multi-line classes will be derived from this class.

The cursor location is specified in the attribute *.edit_cell*. Note that this follows the (odd) curses convention of specifying the row, then the column.

values should be specified as a two-dimensional array.

The convenience function *set_grid_values_from_flat_list(new_values, max_cols=None, reset_cursor=True)* takes a flat list and displays it on the grid.

The following arguments can be passed to the constructor:

```
columns = None
column_width = None,
col_margin=1,
row_height = 1,
values = None
always_show_cursor = False
select_whole_line = False (new in version 4.2)
```

Classes derived from SimpleGrid may wish to modify the following class attributes:

```
_contained_widgets    = textbox.Textfield
default_column_number = 4
additional_y_offset   = 0  # additional offset to leave within the widget before␣
↪the grid
additional_x_offset   = 0  # additional offset to leave within the widget before␣
↪the grid
select_whole_line   # highlight the whole line that the cursor is on
```

GridColTitles Like the simple grid, but uses the first two lines of the display to display the column titles. These can be provided as a *col_titles* argument at the time of construction, or by setting the *col_titles* attribute at any time.

In either case, provide a list of strings.

Customizing the appearance of individual grid cells

New in version 4.8.2.

For some applications it may be desirable to customize the attributes of the contained grid widgets depending upon their content. Grid widgets call a method called *custom_print_cell(actual_cell, display_value)* after they have set the value of a cell and before the content of the cell is drawn to the screen. The parameter *actual_cell* is the underlying widget object being used for display, while *display_value* is the object that has been set as the content of the cell (which is the output of the *display_value* method).

The following code demonstrates how to use this facility to adjust the color of the text displayed in a grid. My thanks are due to Johan Lundstr?m for suggesting this feature:

```
class MyGrid(npyscreen.GridColTitles):
    # You need to override custom_print_cell to manipulate how
    # a cell is printed. In this example we change the color of the
    # text depending on the string value of cell.
    def custom_print_cell(self, actual_cell, cell_display_value):
        if cell_display_value =='FAIL':
           actual_cell.color = 'DANGER'
        elif cell_display_value == 'PASS':
           actual_cell.color = 'GOOD'
        else:
           actual_cell.color = 'DEFAULT'

def myFunction(*args):
    # making an example Form
    F = npyscreen.Form(name='Example viewer')
    myFW = F.add(npyscreen.TitleText)
    gd = F.add(MyGrid)

    # Adding values to the Grid, this code just randomly
    # fills a 2 x 4 grid with random PASS/FAIL strings.
    gd.values = []
    for x in range(2):
        row = []
        for y in range(4):
            if bool(random.getrandbits(1)):
                row.append("PASS")
            else:
                row.append("FAIL")
        gd.values.append(row)
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
```

CHAPTER 12

Widgets: Other Controls

Checkbox, RoundCheckBox These offer a single option - the label is generated from the attribute *name*, as for titled widgets. The attribute *value* is either true or false.

The function whenToggled(self) is called when the user toggles the state of the checkbox. You can overload it.

CheckboxBare This has no label, and is only useful in special circumstances. It was added at user request.

CheckBoxMultiline, RoundCheckBoxMultiline This widgets allow the label of the checkbox to be more than one line long. The name of the widget should be specified as a list or tuple of strings.

To use these widgets as part of a multiline widget, do the following:

```
class MultiSelectWidgetOfSomeKind(npyscreen.MultiSelect):
    _contained_widgets = npyscreen.CheckBoxMultiline
    _contained_widget_height = 2

    def display_value(self, vl):
        # this function should return a list of strings.
```

New in version 2.0pre83.

Button Functionally similar to the Checkbox widgets, but looking different. The Button is usually used for OK and Cancel Buttons on Forms and similar things, though they should probably be replaced with the ButtonPress type. The colour that the button is shown when selected is either an inverse of the colour of the button, or else selected by the attribute *cursor_color*. This value can also be passed in to the constructor. If this value is None, the inverse of the button colour will be used.

ButtonPress Not a toggle, but a control. This widget has the method *whenPressed(self)*, which you should overload to do your own things.

From version 4.3.0 onwards, the constructor accepts an argument *when_pressed_function=None*. If a callable is specified in this way, it will be called instead of the method *whenPressed*. NB. The when_pressed_function functionality is potentially dangerous. It can set up a circular reference that the garbage collector will never free. If this is a risk for your program, it is best to subclass this object and override the *when_pressed_function* method instead.

FormControlCheckbox A common use of Checkbox is to offer the user the option to enter additional data. For example "Enter Expiry Date". In such a case, the Form needs to display additional fields in some cases, but not in others. FormControlCheckbox makes this trivial.

Two methods are defined:

addVisibleWhenSelected(*wg*) *wg* should be a widget.

This method does not create a widget, but instead puts an existing widget under the control of the FormControlCheckbox. If FormControlCheckbox is selected, the widget will be visible.

As many widgets as you wish can be added in this way.

addInvisibleWhenSelected(*wg*) Widgets registered in this way are visible only when the FormControlCheckbox is not selected.

AnnotateTextboxBase, TreeLineAnnotated, TreeLineSelectableAnnotated The *AnnotateTextboxBase* class is mainly intended for use by the multiline listing widgets, for situations where each item displayed needs an annotation supplied to the left of the entry itself. The API for these classes is slightly ugly, because these classes were originally intended for internal use only. It is likely that more user-friendly versions will be supplied in a later release. Classes derived from *AnnotateTextboxBase* should define the following:

ANNOTATE_WIDTH This class attribute defines how much margin to leave before the text entry widget itself. In the TreeLineAnnotated class the margin needed is calculated dynamically, and ANNOTATE_WIDTH is not needed.

getAnnotationAndColor This function should return a tuple consisting of the string to display as the annotation and the name of the colour to use when displaying it. The colour will be ignored on B/W displays, but should be provided in all cases, and the string should not be longer than *ANNOTATE_WIDTH*, although by default the class does not check this.

annotationColor*, *annotationNoColor These methods draw the annotation on the screen. If using strings only, these should not need overriding. If one is altered, the other should be too, since npyscreen will use one if the display is configured for colour and the other if configured for black and white.

CHAPTER 13

Widgets: Titled Widgets

Most versions of the standard widget set come in two forms - a basic version and a corresponding version that also prints a label with the name of the widget. For example, Textfield and TitleText.

The Title versions are in fact a wrapper around a contained widget, rather than being a proper widget in their own right, and this can be a cause of confusion when modifying their behaviour.

In general, to create your own version of these widgets, you should first create the contained widget, and then create a titled version.

For example:

```
class NewTextWidget(textbox.Textfield):
        # all of the custom code for this class
        # should go here.


class TitleProductSearch(TitleText):
        _entry_type = NewTextWidget
```

You can adjust where the child widget is placed on the screen by passing in the argument *begin_entry_at* to the constructor. The default is 16. You can also override whether the widget uses a separate line for the title by passing in the argument *use_two_lines=True\False* at the time the widget is created. The default *use_two_lines=None* will keep the title and the contained widget on the same line, unless the label is too long.

You can change the label color at creation time using the argument *labelColor='LABEL'*. You can specify any of the color names from the theme you are using.

After creation, the two widgets managed by the TitleWidget can be accessed through the *label_widget* and *entry_widget* attributes of the object.

Titled multiline widgets

If you are creating titled versions of the multiline widgets, you will find it better to inherit from the class *TitleMultiLine* instead, which wraps more of the multiline functionality.

CHAPTER 14

Widgets: Box Widgets

These widgets work in a similar way to the Titled versions of widgets. The box widget contains a widget of another class.

BoxBasic BoxBasic prints a box with an optional name and footer on the screen. It is intended as a base class for further widgets, not for direct use.

BoxTitle BoxTitle is a hybrid of the Title widget and the Multiline widget. Again, it is mostly intended as a base class for more complex layouts. This class has a *_contained_widget* attribute that puts a widget inside the box when the class is created. In the Boxtitle class this is a Multiline widget. The title of the widget can be passed to *__init__* the parameter *name*=..... Another perimeter *footer*=... gives the text for the footer of the box. These correspond to attributes named *name* and *footer* which can be changed at any time.

The attribute *entry_widget* gives direct access to the contained widget.

The properties *editable*, *values*, and *value* give direct access to the attributes of *entry_widget*.

The constructor for this widget can be passed the argument *contained_widget_arguments*. This should be a dictionary of arguments that will be passed to the entry_widget when it is created. Note that no sanity checking is done on this dictionary at this time. (New in version 4.8.0)

Your own versions of these widgets can be created in the same way as new Titled widgets. Create the contained widget class first, and then create the box class wrapper class:

```
class NewMultiLineClass
        # Do all sorts of clever things here!
        # ....

 class BoxTitle(BoxBasic):
     _contained_widget = NewMultiLineClass
```

CHAPTER 15

All about Key Bindings

What is going on

Many objects can take actions based on user key presses. All such objects inherit from the internal class InputHandler. That class defines a dictionary called *handlers* and a list called *complex_handlers*. Both of these are set up by a method called *set_up_handlers* called during the Constructor.

handlers Might look something like this:

```
{curses.ascii.NL:   self.h_exit_down,
 curses.ascii.CR:   self.h_exit_down,
 curses.ascii.TAB:  self.h_exit_down,
 curses.KEY_DOWN:   self.h_exit_down,
 curses.KEY_UP:     self.h_exit_up,
 curses.KEY_LEFT:   self.h_exit_left,
 curses.KEY_RIGHT:  self.h_exit_right,
 "^P":              self.h_exit_up,
 "^N":              self.h_exit_down,
 curses.ascii.ESC:  self.h_exit_escape,
 curses.KEY_MOUSE:  self.h_exit_mouse,
 }
```

If a key is pressed (note support for notations like "^N" for "Control-N" and "!a" for "Alt N") that exists as a key in this dictionary, the function associated with it is called. No further action is taken. By convention functions that handle user input are prefixed with h_.

complex_handlers This list should contain list or tuple pairs like this (test_func, dispatch_func).

If the key is not named in the dictionary *handlers*, each test_func is run. If it returns True, the dispatch_func is run and the search stops.

Complex handlers are used, for example, to ensure that only printable characters are entered into a textbox. Since they will be run frequently, there should be as few of them as possible, and they should execute as quickly as possible.

When a user is editing a widget and a key is pressed, *handlers* and then *complex_handlers* are used to try to find a function to execute. If the widget doesn't define an action to be taken, the *handlers* and *complex_handlers* of the parent Form are then checked. Consequently, if you want to override the handler for a key that is already bound, like the Enter key, keep in mind that you do so on Widgets and not on the Form they're part of, as the Widget's handlers take precedence.

Adding your own handlers

Objects that can handle user input define the following methods to assist with adding your own key bindings:

add_handlers(new_handlers) *new_handlers* should be a dictionary.

add_complex_handlers(new_handlers) *new_handlers* should be a list of lists. Each sublist must be a pair *(test_function, callback)*

CHAPTER 16

Enhancing Mouse Support

Widgets that wish to handle mouse events in more detail should override the method *.handle_mouse_event(self, mouse_event)*. Note that *mouse_event* is a tuple:

```
def handle_mouse_event(self, mouse_event):
    mouse_id, x, y, z, bstate = mouse_event # see note below.
    # Do things here....
```

This is mostly useful, but x and y are absolute positions, rather than relative ones. For that reason, you should use the convenience function provided to convert these values into co-ordinates relative to the widget. Thus, most mouse handling functions will look like this:

```
def handle_mouse_event(self, mouse_event):
    mouse_id, rel_x, rel_y, z, bstate = self.interpret_mouse_event(mouse_event)
    # Do things here.....
```

The mouse handler will only be called if the widget is "editable". In very rare cases, you may wish to have a non-editable widget respond to mouse events. In that case, you can set the widget's attribute *.self.interested_in_mouse_even_when_not_editable* to True.

See the Python Library curses module documentation for more detail on mouse events.

CHAPTER 17

Support for Colour

Setting up colour

All of the standard widgets are entirely usable on a monochrome terminal. However, it's a colourful world these days, and npyscreen lets you display your widgets in, well, if not Technicolor(TM) then as close as curses will allow.

Colour is handled by the ThemeManager class. Generally, your application should stick to using one ThemeManager, which you should set using the *setTheme(ThemeManager)* function. So for example:

```
npyscreen.setTheme(npyscreen.Themes.ColorfulTheme)
```

Any default themes defined by npyscreen will be accessible via npyscreen.Themes.

A basic theme looks like this:

```
class DefaultTheme(npyscreen.ThemeManager):
    default_colors = {
    'DEFAULT'     : 'WHITE_BLACK',
    'FORMDEFAULT' : 'WHITE_BLACK',
    'NO_EDIT'     : 'BLUE_BLACK',
    'STANDOUT'    : 'CYAN_BLACK',
    'CURSOR'      : 'WHITE_BLACK',
    'CURSOR_INVERSE': 'BLACK_WHITE',
    'LABEL'       : 'GREEN_BLACK',
    'LABELBOLD'   : 'WHITE_BLACK',
    'CONTROL'     : 'YELLOW_BLACK',
    'IMPORTANT'   : 'GREEN_BLACK',
    'SAFE'        : 'GREEN_BLACK',
    'WARNING'     : 'YELLOW_BLACK',
    'DANGER'      : 'RED_BLACK',
    'CRITICAL'    : 'BLACK_RED',
    'GOOD'        : 'GREEN_BLACK',
    'GOODHL'      : 'GREEN_BLACK',
    'VERYGOOD'    : 'BLACK_GREEN',
    'CAUTION'     : 'YELLOW_BLACK',
```

```
    'CAUTIONHL'    : 'BLACK_YELLOW',
    }
```

The colours - such as WHITE_BLACK ("white on black") - are defined in the *initialize_pairs* method of the ThemeManager class. The following are defined by default:

```
('BLACK_WHITE',       curses.COLOR_BLACK,       curses.COLOR_WHITE),
 ('BLUE_BLACK',        curses.COLOR_BLUE,        curses.COLOR_BLACK),
 ('CYAN_BLACK',        curses.COLOR_CYAN,        curses.COLOR_BLACK),
 ('GREEN_BLACK',       curses.COLOR_GREEN,       curses.COLOR_BLACK),
 ('MAGENTA_BLACK',     curses.COLOR_MAGENTA,     curses.COLOR_BLACK),
 ('RED_BLACK',         curses.COLOR_RED,         curses.COLOR_BLACK),
 ('YELLOW_BLACK',      curses.COLOR_YELLOW,      curses.COLOR_BLACK),
)
```

('WHITE_BLACK' is always defined.)

If you find you need more, subclass ThemeManager and change class attribute *_colours_to_define*. You are able to use colours other than the standard curses ones, but since not all terminals support doing so, npyscreen does not by default.

If you want to disable all colour in your application, npyscreen defines two convenient functions: *disableColor()* and *enableColor()*.

How Widgets use colour

When a widget is being drawn, it asks the active ThemeManager to tell it appropriate colours. 'LABEL', for example, is a label given to colours that will be used for the labels of widgets. The Theme manager looks up the relevant name in its *default_colors* dictionary and returns the appropriate colour-pair as an curses attribute that is then used to draw the widget on the screen.

Individual widgets often have *color* attribute of their own (which may be set by the constructor). This is usually set to 'DEFAULT', but could be changed to any other defined name. This mechanism typically only allows individual widgets to have one particular part of their colour-scheme changed.

Title... versions of widgets also define the attribute *labelColor*, which can be used to change the colour of their label colour.

Defining custom colours (strongly discouraged)

On some terminals, it is possible to define custom colour values. rxvt/urxvt is one such terminal. From version 4.8.4 onwards, support for this is built in to theme manager classes.

The class variable color_values will be used when the class is initialized to redefine custom colour values:

```
_color_values = (
                # redefining a standard color
        (curses.COLOR_GREEN, (150,250,100)),
                # defining another color
                (70, (150,250,100)),
    )
```

NB. Current versions of npyscreen make no effort to reset these values when the application exits.

Use of this facility is discouraged, because it is impossible to tell reliably whether or not a terminal actually supports custom colours. This feature was added at user request to support a custom application.

CHAPTER 18

Displaying Brief Messages and Choices

The following functions allow you to display a brief message or choice to the user.

Notify and related methods are implemented in `npyscreen/utilNotify.py`

The examples on this page build from this basic program:

```
import npyscreen


class NotifyBaseExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.format(key_
↪of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add(npyscreen.FixedText, value=what_to_display)

    def exit_application(self):
        self.parentApp.setNextForm(None)
        self.editing = False


class MyApplication(npyscreen.NPSAppManaged):
    def onStart(self):
        self.addForm('MAIN', NotifyBaseExample, name='To be improved upon')


if __name__ == '__main__':
    TestApp = MyApplication().run()
```

notify(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*)
This function displays a message on the screen. It does not block and the user cannot interact with it - use it to display messages like "Please Wait" while other things are happening.

Listing 18.1: ../examples/notify/notify.py snippet

```
import npyscreen
import time


class NotifyExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.spawn_notify_popup})
        self.add(npyscreen.FixedText, value=what_to_display)

    def spawn_notify_popup(self, code_of_key_pressed):
        message_to_display = 'I popped up \n passed: {}'.format(code_of_key_
↪pressed)
        npyscreen.notify(message_to_display, title='Popup Title')
        time.sleep(1) # needed to have it show up for a visible amount of time
```

notify_wait(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*)
This function displays a message on the screen, and blocks for a brief amount of time. The user cannot interact with it.

Listing 18.2: ../examples/notify/notify_wait.py snippet

```
class NotifyWaitExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.spawn_notify_popup})
        self.add(npyscreen.FixedText, value=what_to_display)

    def spawn_notify_popup(self, code_of_key_pressed):
        message_to_display = 'I popped up \n passed: {}'.format(code_of_key_
↪pressed)
        npyscreen.notify_wait(message_to_display, title='Popup Title')
```

notify_confirm(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*, *editw=0*)
Display a message and an OK button. The user can scroll the message if needed. editw controls which widget is selected when the dialog is first displayed; set to 1 to have the OK button active immediately.

Listing 18.3: ../examples/notify/notify_confirm.py snippet

```
class NotifyConfirmExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

```

```
9           self.add_handlers({key_of_choice: self.spawn_notify_popup})
10          self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
    ↪application
11          self.add(npyscreen.FixedText, value=what_to_display)
12
13      def spawn_notify_popup(self, code_of_key_pressed):
14          message_to_display = 'You need to confirm me, so hit TAB, then ENTER'
15          npyscreen.notify_confirm(message_to_display, title= 'popup')
```

notify_ok_cancel (*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *editw = 0*)
Display a message and return True if the user selected 'OK' and False if the user selected 'Cancel'.

Listing 18.4: ../examples/notify/notify_ok_cancel.py snippet

```
4   class NotifyOkCancelExample(npyscreen.Form):
5       def create(self):
6           key_of_choice = 'p'
7           what_to_display = 'Press {} for popup \n Press escape key to quit'.
    ↪format(key_of_choice)
8
9           self.add_handlers({key_of_choice: self.spawn_notify_popup})
10          self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
    ↪application
11          self.add(npyscreen.FixedText, value=what_to_display)
12
13      def spawn_notify_popup(self, code_of_key_pressed):
14          message_to_display = 'You have a choice, to Cancel and return false, or␣
    ↪Ok and return true.'
15          notify_result = npyscreen.notify_ok_cancel(message_to_display, title=
    ↪'popup')
16          npyscreen.notify_wait('That returned: {}'.format(notify_result), title=
    ↪'results')
```

notify_yes_no (*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *editw = 0*)
Similar to *notify_ok_cancel* except the names of the buttons are 'Yes' and 'No'. Returns True or False.

selectFile (*select_dir=False*, *must_exist=False*, *confirm_if_exists=True*, *sort_by_extension=True*)
Display a dialog box for the user to select a filename. Uses the called from directory as the initial folder. The return value is the name of the file selected.

Warning: This form is currently experimental.

Listing 18.5: ../examples/notify/select_file.py snippet

```
4   class SelectFileExample(npyscreen.Form):
5       def create(self):
6           key_of_choice = 'p'
7           what_to_display = 'Press {} for popup \n Press escape key to quit'.
    ↪format(key_of_choice)
8
9           self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
    ↪application
10          self.add_handlers({key_of_choice: self.spawn_file_dialog})
11          self.add(npyscreen.FixedText, value=what_to_display)
12
13      def spawn_file_dialog(self, code_of_key_pressed):
14          the_selected_file = npyscreen.selectFile()
15          npyscreen.notify_wait('That returned: {}'.format(the_selected_file),␣
    ↪title= 'results')
```

CHAPTER 19

Blanking the Screen

blank_terminal()

This function blanks the terminal. It may sometimes be needed if Forms are being displayed that do not fill the whole screen.

Listing 19.1: ../examples/notify/blank_terminal.py snippet

```
import npyscreen
import time


class BlankTerminalExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'b'
        what_to_display = 'Press {} to blank screen \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.initiate_blanking_sequence})
        self.add(npyscreen.FixedText, value=what_to_display)

    def initiate_blanking_sequence(self, code_of_key_pressed):
        npyscreen.blank_terminal()
        time.sleep(1.5)
        npyscreen.notify('..and we\'re back', title='Phew')
        time.sleep(0.75)

    def exit_application(self):
        self.parentApp.setNextForm(None)
        self.editing = False


class MyApplication(npyscreen.NPSAppManaged):
    def onStart(self):
        self.addForm('MAIN', BlankTerminalExample, name='To show off blank_screen
↪')
```

```
28
29
30 if __name__ == '__main__':
31     TestApp = MyApplication().run()
```

CHAPTER 20

Application Support

Options and Option Lists

One common problem is to display a list of options to the user. In simple applications, a custom designed form may be used for this purpose, but for many tasks an automatically generated form is desirable. An *experimental* system to support this was introduced in version 2.0pre84.

At the core of this system is the concept of the *Option* object. These objects store either single values or lists of values depending on their type, as well as any documentation that goes with the option in question and which should be presented to the user. Options are created with the following arguments: *OptionType(name, value=None, documentation=None, short_explanation=None, option_widget_keywords = None, default = None)*. The *short_explanation* argument is currently not used by the default widgets but will be used in future versions. Option classes designed to allow the user to select from a limited range of options may also be created with the *choices* arguement.

All option classes also have the class attributes DEFAULT and WIDGET_TO_USE. The first of these defines a default value if this is not specified. The second specifies the class of widget that is used to allow the user to adjust the option in question.

The following option classes are currently defined: *OptionFreeText*, *OptionSingleChoice*, *OptionMultiChoice*, *OptionMultiFreeList*, *OptionBoolean*, *OptionFilename*, *OptionDate*, *OptionMultiFreeText*. The value stored in the Option object should be set with the *set(value)* method and retrieved with the *get()* method. All Option classes also define a method *when_set* which can be overridden and which will be called after the value is changed. Options that allow the user to select from a series of limited choices also have the methods *setChoices(choices)* and *getChoices*.

An option list can be displayed using the *OptionListDisplay* widget. This takes a list of options as its *values* attribute. If an option is selected, a form will be presented to the user that will display the documentation (if any) and allow the user to change the value of it.

Option collections may be gathered together with an *OptionList* object. *OptionList* classes have an attribute *options*. This is simply a list, to which Option objects may be appended. Future versions may define a different API. The purpose of *OptionList* objects is to help saving and restoring collections of objects to test files. The format of these files is a custom text format, similar to standard unix file but capable of storing and restoring lists of strings (using tab characters as a seperator.) This format is still evolving, and may be changed in future versions. Only values that vary from the default are stored.

OptionList objects may be created with a *filename* argument, and have the methods *write_to_file(fn=None)* and *reload_from_file(fn=None)*.

The class *SimpleOptionForm* is a form designed to show how these elements work. The *OptionListDisplay* widget is created as an attribute with the name *wOptionList*.

Example Code

The following short demo program will store the specified options in the file '/tmp/test' between invocations:

```
#!/usr/bin/env python
# encoding: utf-8

import npyscreen
class TestApp(npyscreen.NPSApp):
    def main(self):
        Options = npyscreen.OptionList()

        # just for convenience so we don't have to keep writing Options.options
        options = Options.options

        options.append(npyscreen.OptionFreeText('FreeText', value='', documentation=
↪"This is some documentation."))
        options.append(npyscreen.OptionMultiChoice('Multichoice', choices=['Choice 1',
↪ 'Choice 2', 'Choice 3']))
        options.append(npyscreen.OptionFilename('Filename', ))
        options.append(npyscreen.OptionDate('Date', ))
        options.append(npyscreen.OptionMultiFreeText('Multiline Text', value=''))
        options.append(npyscreen.OptionMultiFreeList('Multiline List'))

                try:
                Options.reload_from_file('/tmp/test')
        except FileNotFoundError:
                        pass

        F  = npyscreen.Form(name = "Welcome to Npyscreen",)

        ms = F.add(npyscreen.OptionListDisplay, name="Option List",
                values = options,
                scroll_exit=True,
                max_height=None)

        F.edit()

        Options.write_to_file('/tmp/test')

if __name__ == "__main__":
    App = TestApp()
    App.run()
```

CHAPTER 21

Writing More Complex Forms

A very typical style of programming for terminal applications has been to have a screen that has a command line, typically at the bottom of the screen, and then some kind of list widget or other display taking up most of the screen, with a title bar at the top and a status bar above the command line. Variations on this scheme are found in applications like Mutt, less, Vim, irssi and so on.

To make writing these kinds of form easier, npyscreen provides a series of classes that are intended to work together.

FormMuttActive, FormMuttActiveWithMenus, FormMuttActiveTraditional, FormMuttActiveTraditionalWithMenus

These classes define the basic form. The following *class attributes* dictate exactly how the form is created:

```
MAIN_WIDGET_CLASS   = wgmultiline.MultiLine
MAIN_WIDGET_CLASS_START_LINE = 1
STATUS_WIDGET_CLASS = wgtextbox.Textfield
STATUS_WIDGET_X_OFFSET = 0
COMMAND_WIDGET_CLASS= wgtextbox.Textfield
COMMAND_WIDGET_NAME = None
COMMAND_WIDGET_BEGIN_ENTRY_AT = None
COMMAND_ALLOW_OVERRIDE_BEGIN_ENTRY_AT = True

DATA_CONTROLER    = npysNPSFilteredData.NPSFilteredDataList

ACTION_CONTROLLER  = ActionControllerSimple
```

The default definition makes the following instance attributes available after initalization:

```
# Widgets -
self.wStatus1 # by default a title bar
self.wStatus2 # just above the command line
self.wMain    # the main area of the form - by default a MultiLine object
self.wCommand # the command widget

self.action_controller # not a widget. See below.
```

The form's *.value* attribute is set to an instance of the object specified by DATA_CONTROLLER.

Typically, and application will want to define its own DATA_CONTROLLER and ACTION_CONTROLLER.

The difference between the traditional and non-traditional forms is that in the traditional form, the focus stays always with the command line widget, although some keypresses will be passed to the MAIN_WIDGET_CLASS - so that, from the user's point of view, it looks as if he/she is interacting with both at once.

TextCommandBox The TextCommandBox is like a usual text box, except that it passes what the user types to the action_controller. In addition, it can keep a history of the commands entered. See the documentation on ActionControllerSimple for more details.

TextCommandBoxTraditional This is the same as the TextCommandBox, except that it additionally will pass certain keystrokes to the widget specified by *self.linked_widget*. In the default case, any keystroke that does not match a handler in TextCommandBoxTraditional will be passed to the linked widget. Additionally, any keystroke that is listed in the list *self.always_pass_to_linked_widget* will be handled by the linked widget. However, if the current command line begins with any character that is listed in the class attribute *BEGINNING_OF_COMMAND_LINE_CHARS*, the user input will be handled by this class, not by the linked widget.

This is rather complicated, but an example will make it clearer. The default BEGINNING_OF_COMMAND_LINE_CHARS specifies that ':' or '/' marks the beginning of a command. After that point, keypresses are handled by this widget, not by the linked widget, so that the up and down arrows start to navigate the command history. However, if the command line is currently empty, those keys navigate instead the linked widget.

As in the TextCommandBox widget, the value of the command line is passed to the parent form's action_controller object.

ActionControllerSimple This object receives command lines and executes call-back functions.

It recognises two types of command line - a "live" command line, where an action is taken with every change in the command line, and a command that is executed when the return key is pressed.

Callbacks are added using the *add_action(ident, function, live)*, method. 'ident' is a regular expression that will be matched against the command line, *function* is the callback itself and *live* is either True or False, to specify whether the callback should be executed with every keypress (assuming that 'ident' matches).

Command lines that match the regular expression 'ident' cause the call-back to be called with the following arguments: *call_back(command_line, control_widget_proxy, live=True)*. Here *command_line* is the string that is the command line, *control_widget_proxy* is a weak reference to the command line widget, and live specifies whether the function is being called 'live' or as a result of a return.

The method *create()* can be overridden. It is called when the object is created. The default does nothing. You probably want to use this as a place to call *self.add_action*.

NPSFilteredDataBase The default *NPSFilteredDataBase* class suggests how the code to manage the display might be separated out into a separate object. The precise methods will be very application dependent. This is not an essential part of this kind of application, but it is good practice to keep the logic of (for example) database access separate from the logic of the user interface.

Example Code

The following example shows how this model works. The application creates an ActionController that has a search action. This action calls the user-defined function set_search, which communicates with the Form's parent.value (actually a NPSFilteredDataBase class). It then uses this class to set the values in wMain.values and calls wMain.display() to update the display.

FmSearchActive is simply a FormMuttActiveTraditional class, with a class attribute that specifies that the form should use our action controller:

```
class ActionControllerSearch(npyscreen.ActionControllerSimple):
    def create(self):
```

```
        self.add_action('^/.*', self.set_search, True)

    def set_search(self, command_line, widget_proxy, live):
        self.parent.value.set_filter(command_line[1:])
        self.parent.wMain.values = self.parent.value.get()
        self.parent.wMain.display()


class FmSearchActive(npyscreen.FormMuttActiveTraditional):
    ACTION_CONTROLLER = ActionControllerSearch

class TestApp(npyscreen.NPSApp):
    def main(self):
        F = FmSearchActive()
        F.wStatus1.value = "Status Line "
        F.wStatus2.value = "Second Status Line "
        F.value.set_values([str(x) for x in range(500)])
        F.wMain.values = F.value.get()

        F.edit()


if __name__ == "__main__":
    App = TestApp()
    App.run()
```

CHAPTER 22

Writing Tests

(New in version 4.7.0)

It is possible to script npyscreen application keyboard input for the purposes of testing.

The npyscreen module exports the following dictionary containing the relevant settings:

```
TEST_SETTINGS = {
    'TEST_INPUT': None,
    'TEST_INPUT_LOG': [],
    'CONTINUE_AFTER_TEST_INPUT': False,
    'INPUT_GENERATOR': False
    }
```

If 'TEST_INPUT' is None the application progresses normally. If it is an array, keystrokes are loaded from the left hand side of the array and fed to the application in place of getting input from the keyboard. Note that special characters such as *curses.KEYDOWN* can be handled, and control characters can be indicated by a string such as *"^X"*.

A keypress that is fed to the application in this way is automatically appended to *'TEST_INPUT_LOG'*, so it is possible to see where an error occurred when handling input.

If 'CONTINUE_AFTER_TEST_INPUT' is true, then after the automatic input has been specified, *'TEST_INPUT'* is set to *None* and the application continues normally. If it is False, then the exception *ExhaustedTestInput* is raised instead. This would allow a unittest to then test the state of the application.

'INPUT_GENERATOR' can be set to an iterable object. Each keypress will be read by calling *next(INPUT_GENERATOR)*. Provided the iterable object chosen is thread-safe, this makes it easy to use one thread to feed the test input. This can be used in preference to TEST_INPUT. New in Version 4.9 and added at user request.

Convenience Functions (new in version 4.8.5)

npyscreen.**add_test_input_from_iterable**(*iterable*)
Add each item of *iterable* to *TEST_SETTINGS['TEST_INPUT']*.

npyscreen.**add_test_input_ch**(*ch*)
Add *ch* to *TEST_SETTINGS['TEST_INPUT']*.

Preventing Forking for writing unittests

In order to avoid a memory leak in the underlying curses module, the npyscreen library sometimes chooses to run the application code in a forked process. For testing purposes this is usually undesirable, and you probably want to pass *fork=False* to the *run()* method of your application for testing purposes.

Example

The following is a trivial example:

```
#!/usr/bin/python
import curses
import npyscreen

npyscreen.TEST_SETTINGS['TEST_INPUT'] = [ch for ch in 'This is a test']
npyscreen.TEST_SETTINGS['TEST_INPUT'].append(curses.KEY_DOWN)
npyscreen.TEST_SETTINGS['CONTINUE_AFTER_TEST_INPUT'] = True

class TestForm(npyscreen.Form):
    def create(self):
        self.myTitleText = self.add(npyscreen.TitleText, name="Events (Form␣
↪Controlled):", editable=True)

class TestApp(npyscreen.StandardApp):
    def onStart(self):
        self.addForm("MAIN", TestForm)


if __name__ == '__main__':
    A = TestApp()
    A.run(fork=False)
    # 'This is a test' will appear in the first widget, as if typed by the user.
```

CHAPTER 23

Example Application: A simple address book

An address book applicaton requires a database. For the sake of convenience, we will use sqlite.

```
class AddressDatabase(object):
    def __init__(self, filename="example-addressbook.db"):
        self.dbfilename = filename
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute(
        "CREATE TABLE IF NOT EXISTS records\
            ( record_internal_id INTEGER PRIMARY KEY, \
              last_name     TEXT, \
              other_names   TEXT, \
              email_address TEXT \
              )" \
            )
        db.commit()
        c.close()

    def add_record(self, last_name = '', other_names='', email_address=''):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('INSERT INTO records(last_name, other_names, email_address) \
                    VALUES(?,?,?)', (last_name, other_names, email_address))
        db.commit()
        c.close()

    def update_record(self, record_id, last_name = '', other_names='', email_address='
↪'):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('UPDATE records set last_name=?, other_names=?, email_address=? \
                    WHERE record_internal_id=?', (last_name, other_names, email_
↪address, \
                                                        record_id))
        db.commit()
```

```
        c.close()

    def delete_record(self, record_id):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('DELETE FROM records where record_internal_id=?', (record_id,))
        db.commit()
        c.close()

    def list_all_records(self, ):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('SELECT * from records')
        records = c.fetchall()
        c.close()
        return records

    def get_record(self, record_id):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('SELECT * from records WHERE record_internal_id=?', (record_id,))
        records = c.fetchall()
        c.close()
        return records[0]
```

The main screen of the application will be a list of names. When the user selects a name, we will want to edit it. We will subclass MultiLineAction, and override *display value* to change how each record is presented. We will also override the method *actionHighlighted* to switch to the edit form when required. Finally, we will add two new keypresses - one to add and one to delete records. Before switching to the EDITRECORDFM, we either set its value to None, if creating a new form, or else set its value to that of the record we wish to edit.

```
class RecordList(npyscreen.MultiLineAction):
    def __init__(self, *args, **keywords):
        super(RecordList, self).__init__(*args, **keywords)
        self.add_handlers({
            "^A": self.when_add_record,
            "^D": self.when_delete_record
        })

    def display_value(self, vl):
        return "%s, %s" % (vl[1], vl[2])

    def actionHighlighted(self, act_on_this, keypress):
        self.parent.parentApp.getForm('EDITRECORDFM').value =act_on_this[0]
        self.parent.parentApp.switchForm('EDITRECORDFM')

    def when_add_record(self, *args, **keywords):
        self.parent.parentApp.getForm('EDITRECORDFM').value = None
        self.parent.parentApp.switchForm('EDITRECORDFM')

    def when_delete_record(self, *args, **keywords):
        self.parent.parentApp.myDatabase.delete_record(self.values[self.cursor_
↪line][0])
        self.parent.update_list()
```

The actual form to display the record list will be a FormMutt subclass. We will alter the *MAIN_WIDGET_CLASS* class variable to use our RecordList widget, and make sure that the list of records is updated every time the form is

presented to the user.

```
class RecordListDisplay(npyscreen.FormMutt):
    MAIN_WIDGET_CLASS = RecordList
    def beforeEditing(self):
        self.update_list()

    def update_list(self):
        self.wMain.values = self.parentApp.myDatabase.list_all_records()
        self.wMain.display()
```

The form to edit each record will be an example of an ActionForm. Records will only be altered when the user selects the 'ok' button. Before the form is presented to the user, the values of each of the individual widgets are updated to match the database record, or cleared if we are creating a new record.

```
class EditRecord(npyscreen.ActionForm):
    def create(self):
        self.value = None
        self.wgLastName   = self.add(npyscreen.TitleText, name = "Last Name:",)
        self.wgOtherNames = self.add(npyscreen.TitleText, name = "Other Names:")
        self.wgEmail      = self.add(npyscreen.TitleText, name = "Email:")

    def beforeEditing(self):
        if self.value:
            record = self.parentApp.myDatabase.get_record(self.value)
            self.name = "Record id : %s" % record[0]
            self.record_id          = record[0]
            self.wgLastName.value   = record[1]
            self.wgOtherNames.value = record[2]
            self.wgEmail.value      = record[3]
        else:
            self.name = "New Record"
            self.record_id          = ''
            self.wgLastName.value   = ''
            self.wgOtherNames.value = ''
            self.wgEmail.value      = ''

    def on_ok(self):
        if self.record_id: # We are editing an existing record
            self.parentApp.myDatabase.update_record(self.record_id,
                                        last_name=self.wgLastName.value,
                                        other_names = self.wgOtherNames.value,
                                        email_address = self.wgEmail.value,
                                        )
        else: # We are adding a new record.
            self.parentApp.myDatabase.add_record(last_name=self.wgLastName.value,
            other_names = self.wgOtherNames.value,
            email_address = self.wgEmail.value,
            )
        self.parentApp.switchFormPrevious()

    def on_cancel(self):
        self.parentApp.switchFormPrevious()
```

Finally, we need an application object that manages the two forms and the database:

```
class AddressBookApplication(npyscreen.NPSAppManaged):
    def onStart(self):
```

```
        self.myDatabase = AddressDatabase()
        self.addForm("MAIN", RecordListDisplay)
        self.addForm("EDITRECORDFM", EditRecord)

if __name__ == '__main__':
    myApp = AddressBookApplication()
    myApp.run()
```

CHAPTER 24

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

K

M

N

O

P

R

S

T

W