PyBrex Documentation

Joshua

January 2025

Contents

PyBrex

PyBrex is a Python library for creating custom LibreOffice based applications using frames and dialogs.

Requirements

1. **LibreOffice**
   - Download and install from LibreOffice.org
2. **LibrePy**
   - A LibreOffice-based IDE for Python development
   - Contact Timothy Hoover (tim@timtech.io) to obtain the latest version of LibrePy

Getting Started

The recommended way to develop PyBrex applications is using LibrePy IDE. After installing the prerequisites above:

1. **Get PyBrex**
   - Create a dedicated directory for PyBrex source code
   - Clone the repository: `git clone ssh://git@168.73.52.35:587/home/git/pybrex.git`
   - Always run `git pull` before starting a new project
2. **Create a LibrePy Project**
   - Create a new project in LibrePy IDE
   - Use 'Embedded project' option
   - Enable "Use the LibrePy executable"
   - Configure LibreOffice instance settings
3. **Copy PyBrex Files**

```
# Replace {workspace} and {project} with your paths
cp -r ~/Documents/pybrex_source/pybrex/source/* ~/{workspace}/{project}/source/
```

4. **Start Development**
   - Check the `examples` directory for implementation patterns
   - Follow the getting started guide in the `docs` directory

Documentation

For detailed information, please refer to: - Getting Started Guide: `docs/getting_started.md` - Example Applications: `examples/` - More Documentation: `docs/`

License

[Insert your license information here]

Welcome to PyBrex!

Welcome to PyBrex, a Python library that enables you to create custom LibreOffice based applications using frames and dialogs. This guide will walk you through the key features and help you start building your first PyBrex application. Before continuing, please ensure you have LibreOffice and LibrePy installed on your system, and take a moment to review the README file for important setup information.

Getting Started

Setting Up PyBrex

Before you can start developing with PyBrex, you'll need to set up your development environment. Follow these steps carefully:

1. **Get PyBrex from GitHub**
   - Create a dedicated directory for PyBrex source code:

     ```
     mkdir ~/Documents/pybrex_source
     ```

   - Navigate to the directory you created:

     ```
     cd ~/Documents/pybrex_source
     ```

   - Clone the PyBrex repository:

     ```
     git clone ssh://git@168.73.52.35:587/home/git/pybrex.git
     ```

   - The repository will be cloned into a new 'pybrex' subdirectory. To update an existing clone later:

     ```
     cd ~/Documents/pybrex_source/pybrex
     git pull
     ```

   - **Important:** Always run `git pull` before starting a new project to ensure you have the latest changes and features
2. **Create a New LibrePy Project**

   Launch LibrePy IDE and create a new project:

   a. Open Project Menu:
   - Select "Project -> New" from the menu bar

   b. Configure Project Properties:
   - Enter a meaningful project name (e.g., "ContactList") - IMPORTANT: Do not use spaces in the project name
   - Select 'Embedded project' (default option)
   - Enable "Use the LibrePy executable (recommended)"

   c. Configure Instance Settings:
   - Choose 'Existing instance' under 'Project instance'
   - Verify the LibreOffice configuration path
     - Default location: `/home/default/.config/libreoffice/4/user` (You may need to change the word **default** to your user name)
     - Adjust path according to your system if different

   d. Additional Configuration:
   - Development tab: Keep default settings
   - Compiler tab: Set 'Reload count' to 5 or higher
   - Click 'OK' to create the project
3. **Copy PyBrex Files**
   - Copy the PyBrex source files into your project's source directory:

     ```
     # Replace {workspace} with your LibrePy workspace directory
     # Replace {project} with your project name
     cp -r ~/Documents/pybrex_source/pybrex/source/* ~/{workspace}/{project}/source/

     # For example, with default paths:
     cp -r ~/Documents/pybrex_source/pybrex/source/* ~/Documents/librepy_workspace/ContactList/sou
     ```

   - Verify the copy was successful by checking that the pybrex directory exists in your project's source folder
   - The pybrex directory should contain subdirectories like dialogs/, docs/, examples/, etc.

Your project in LibrePy is now ready for PyBrex development. The next section will guide you through creating your first PyBrex application.

Creating Your First PyBrex Application

Now that we have our project set up, let's build our first PyBrex application - a simple contact list manager. This example will help you learn the basics of PyBrex development while creating something useful.

First, open LibrePy IDE where we'll start building our application.

1. Create a new directory in your project called `contact_list`
2. Create a new Python file called `main.py` in the `contact_list` directory. This file will contain the main application class and entry point.
3. Create a new Python file called `frame_manager.py` in the `contact_list` directory. This file will contain the code for managing the main frame(or window) of your application.

Your directory structure should now look something like this:

```
ContactList/
├── contact_list/
│   ├── main.py
│   └── frame_manager.py
├── pybrex/
│   ├── dialogs/
│   ├── docs/
│   ├── examples/
│   ├── __init__.py
│   └── ...
└── main.py
```

Creating the Main Window

The `BaseFrame` class is the foundation for creating LibreOffice window applications in PyBrex. Let's create a basic application using BaseFrame:

1. **Create a Frame Manager Class** Let's head to our `frame_manager.py` file and create a class that manages a BaseFrame instance:

```python
import traceback
import uno
from librepy.pybrex import base_frame

class FrameManager:
    '''Frame manager for the contact list application'''

    def __init__(self, parent, ctx, smgr, ps, **kwargs):
        self.parent = parent
        self.ctx = ctx
        self.smgr = smgr
        self.logger = parent.logger
        self.logger.info("FrameManager initialized")

        # Create the base frame with this manager as the parent
        self.base_frame = base_frame.BaseFrame(
            parent=self,
            ctx=ctx,
            smgr=smgr,
            title="Contact List",
            frame_name="contact_list_frame",
            ps=ps,  # (x, y, width, height)
            **kwargs
        )

        self.base_frame.show()

    @property
```

```
    def window(self):
        """Access to the underlying window"""
        return self.base_frame.window

    @property
    def frame(self):
        """Access to the underlying frame"""
        return self.base_frame.frame

    def window_resizing(self, width, height):
        """Handle window resize events"""
        self.logger.info(f"Window resized to {width}x{height}")
        self.parent.window_resizing(width, height)

    def can_close(self):
        """Handle window close request"""
        self.logger.info("Window close request received")
        return True

    def save_window_geometry(self):
        """Save window position and size before closing"""
        self.logger.info("Saving window geometry")
        # You can implement custom geometry saving here
        # For example, save to a config file or database
        pass

    def window_closing(self):
        """Called when window is about to close"""
        self.logger.info("Window is closing")
        # Perform any cleanup needed before window closes
        self.save_window_geometry()

    def dispose(self):
        """Clean up window resources"""
        try:
            self.logger.info("Cleaning up FrameManager resources...")
            # Add cleanup tasks here, such as:
            # - Close database connections
            # - Save application state
            # - Clean up custom UI components
        except Exception as e:
            self.logger.error(f"Error during cleanup: {e}")
            self.logger.error(traceback.format_exc())
```

2. **Create the Main Application Class** In your `main.py` file, create the main application class:

```
import uno
import traceback

class ContactList(object):
    """Contact list application"""

    def __init__(self):

        ctx = uno.getComponentContext()
        smgr = ctx.getServiceManager()

        # Window configuration
        self.ps = (0, 0, 700, 400)  # (x, y, width, height)
        self.menubar_height = 25
        self.toolbar_height = 55
```

```
        # Create frame manager
        from librepy.contact_list import frame_manager
        self.frame_manager = frame_manager.FrameManager(
            self,
            ctx,
            smgr,
            ps=self.ps,
            menubar_height=self.menubar_height
        )

    def window_resizing(self, width, height):
        """Handle window resize events"""
        # Update UI components when window is resized
        if hasattr(self, 'toolbar_manager'):
            self.toolbar_manager.resize(width, height) # We will create this ahead

    def dispose(self):
        """Dispose of the application"""
        try:
            self.frame_manager.dispose()
        except:
            self.logger.error(traceback.format_exc())

def main(*args):
    """Main function for the contact list application"""
    contact_list = ContactList()
```

The BaseFrame class provides several methods that your manager class can implement to handle window events:

- `window_resizing(width, height)`: Called when the window is resized
- `can_close()`: Called before window closes, return False to prevent closing
- `save_window_geometry()`: Called to save window position/size before closing
- `window_closing()`: Called when window is about to close
- `dispose()`: Called to clean up resources

The BaseFrame instance will automatically call these methods on your manager class when the corresponding events occur.

Important Notes: - The `parent` parameter enables parent-child communication between windows - Window geometry is automatically validated before saving - The frame name is automatically made unique to prevent conflicts - Cleanup is handled hierarchically through the dispose chain

Creating a Menubar

PyBrex provides a convenient menubar system that allows you to create professional-looking menus with icons, keyboard shortcuts, and nested submenus. Let's add a menubar to our contact list application.

1. **Create the Menubar Manager** Create a new file called `menubar_manager.py` in your `contact_list` directory:

```
from librepy.pybrex import menubar

class MenubarManager(object):
    def __init__(self, parent, ctx, smgr, frame):
        self.parent = parent
        self.ctx = ctx
        self.smgr = smgr
        self.frame = frame
        self.logger = parent.logger
        self.logger.info("MenubarManager initialized")
        self.menubar = self.create_menubar()

    def create_menubar(self):
        """Menubar for the contact list application"""
```

```
        # Define menu structure using Menu and SubMenu classes
        m = menubar.Menu      # For top-level menus
        sm = menubar.SubMenu  # For menu items and submenus

        menulist = [
            m(0, '~File', None, (
                sm(0, '~New Contact', 'f_new', graphic='document-new.png', key='Ctr N'),
                sm(1, '~Open Contact', 'f_open', graphic='document-open.png', key='Ctr O'),
                sm(None, 'Divider'),
                sm(2, '~Export Contacts', 'f_export', graphic='document-save.png'),
                sm(None, 'Divider'),
                sm(3, '~Settings', 'p_settings', graphic='document-properties.png')
            )),
            m(1, '~Help', None, (
                sm(0, '~About', 'h_about', graphic='help-about.png'),
            )),
        ]

        # Map menu commands to handler functions
        fn = {
            'f_new': self.new_contact,
            'f_open': self.open_contact,
            'f_export': self.export_contacts,
            'p_settings': self.settings,
            'h_about': self.show_about
        }

        return menubar.Menubar(self.parent, self.ctx, self.smgr, self.frame, menulist, fn)

    def dispose(self):
        """Dispose of the menubar manager"""
        try:
            self.menubar.dispose()
        except Exception as e:
            self.logger.error(f"Error during cleanup: {e}")
            self.logger.error(traceback.format_exc())

    # Menubar action handlers
    def new_contact(self, *args): pass
    def open_contact(self, *args): pass
    def export_contacts(self, *args): pass
    def settings(self, *args): pass
    def show_about(self, *args): pass
```

2. **Connect the Menubar** In your main application class (`main.py`), initialize the menubar manager:

```
# Inside ContactList.main
from librepy.contact_list import menubar_manager
self.menubar_manager = menubar_manager.MenubarManager(
    self,
    ctx,
    smgr,
    self.frame_manager
)
```

Menu Configuration Options

The menubar system uses two main classes for defining menu structure:

1. **Menu Class** - For top-level menus:

`Menu(id, name, cmd=None, submenu=None)`

- `id`: Unique numeric identifier
- `name`: Display name (use ~ to underscore the letter following it.)
- `cmd`: Command identifier (usually None for top-level)
- `submenu`: Tuple of SubMenu items

2. **SubMenu Class** - For menu items and submenus:

`SubMenu(id, name, cmd=None, style=0, graphic=None, key=None, submenu=None)`

- `id`: Unique numeric identifier (None for separators)
- `name`: Display name (use ~ to underscore the letter following it.)
- `cmd`: Command identifier for action handling
- `style`: Menu item style (CHECKABLE, AUTOCHECK, RADIOCHECK)
- `graphic`: Icon filename (must be in TOOLBAR_GRAPHICS_DIR)
- `key`: Keyboard shortcut (e.g., 'Ctr N', 'Shift F1')
- `submenu`: Nested submenu items (if any)

Important Notes:

- Menu icons must be placed in the `TOOLBAR_GRAPHICS_DIR` directory
- Use the ~ character to create Alt+letter keyboard shortcuts
- Use `None` as the id to create menu separators
- The `key` parameter supports modifiers:
  - `Ctr`: Control key
  - `Shift`: Shift key
  - `Alt`: Alt key
- Handler functions receive two parameters:
  - `cmd`: The command identifier
  - `checked`: Boolean state for checkable items

Cleanup

Don't forget to add cleanup code in your application's dispose method:

```
def dispose(self):
    """Dispose of the application"""
    try:
        self.frame_manager.dispose()
    except:
        self.logger.error(traceback.format_exc())

    try:
        self.menubar_manager.dispose()
    except:
        self.logger.error(traceback.format_exc())
```

When you run the application now, you should see a window with a fully functional menubar including icons and keyboard shortcuts.

Connect to the main function

Let's head to our `main.py` file and connect to the main function.

```
from librepy.contact_list import ContactList

 contact_list = ContactList()
```

Run the application

Ensure the project is connected, compile and click on start in LibrePy. You should see a new window appear with the title "Contact List".

Congratulations! You've just created your first PyBrex application. BaseFrame has many more features and options that you can use to create more complex applications. For further information on how BaseFrame works, please refer to the BaseFrame documentation.

Next Steps

We will continue to build on this application in the next section. We will add a toolbar.

Adding a Toolbar

PyBrex provides a flexible toolbar system that allows you to create professional-looking toolbars with icons, tooltips, and custom styling. Let's add a toolbar to our contact list application.

1. **Create the Toolbar Manager** Create a new file called `toolbar_manager.py` in your `contact_list` directory:

```
from librepy.pybrex import toolbar

class ToolbarManager:
    def __init__(self, parent, ctx, smgr, frame, **kwargs):
        self.parent = parent
        self.ctx = ctx
        self.smgr = smgr
        self.frame = frame

        # Configure toolbar appearance
        self.toolbar_config = {
            'height': 55,
            'button_width': 50,
            'button_spacing': 50,
            'colors': {
                'border': 0x000000,      # Black border
                'button_normal': 0xE8E8E8,  # Light gray
                'button_hover': 0xF5F5F5,   # Almost white
                'button_pressed': 0xC0C0C0  # Medium gray
            }
        }

        self.toolbar = self.create_toolbar()

    def create_toolbar(self):
        # Define toolbar items: [Type, Label, Icon, Callback, Tooltip, Width_offset]
        toolbar_list = [
            ['Button', 'New', 'document-new.png', self.new_contact, 'Create new contact', 0],
            ['Button', 'Open', 'document-open.png', self.open_contact, 'Open existing contact', 0],
            ['Line', 'line1'],
            ['Button', 'Export', 'document-save.png', self.export_contacts, 'Export contacts', 15],
        ]

        return toolbar.ToolBar(
            self.parent,
            self.ctx,
            self.smgr,
            self.frame,
            toolbar_list,
            **self.toolbar_config
        )

    def resize(self, width, height):
        """Handle toolbar resize events"""
        self.toolbar.resize(width, height)
```

```
    def dispose(self):
        """Clean up toolbar resources"""
        self.toolbar.dispose()

    # Toolbar action handlers
    def new_contact(self, *args): pass
    def open_contact(self, *args): pass
    def export_contacts(self, *args): pass
```

2. **Update the Main Application** Modify your ContactList class in `main.py` to include the toolbar:

```
def __init__(self):
    ctx = uno.getComponentContext()
    smgr = ctx.getServiceManager()

    # Create frame manager
    from librepy.contact_list import frame_manager
    self.frame_manager = frame_manager.FrameManager(
        self, ctx, smgr, ps=self.ps
    )

    # Create menubar manager
    from librepy.contact_list import menubar_manager
    self.menubar_manager = menubar_manager.MenubarManager(
        self, ctx, smgr, self.frame_manager
    )

    # Create toolbar manager
    from librepy.contact_list import toolbar_manager
    self.toolbar_manager = toolbar_manager.ToolbarManager(
        self, ctx, smgr, self.frame_manager
    )

def window_resizing(self, width, height):
    """Handle window resize events by adjusting toolbar dimensions."""
    if hasattr(self, 'toolbar_manager'):
        self.toolbar_manager.resize(width, height)

def dispose(self):
    """Dispose of the contact list application"""
    try:
        self.frame_manager.dispose()
    except:
        self.logger.error(traceback.format_exc())

    try:
        self.menubar_manager.dispose()
    except:
        self.logger.error(traceback.format_exc())

    try:
        self.toolbar_manager.dispose()
    except:
        self.logger.error(traceback.format_exc())
```

Toolbar Configuration Options

The toolbar system provides several configuration options through a dictionary:

```
toolbar_config = {
    # Basic Layout
```

```
    'height': 55,            # Overall toolbar height
    'button_width': 50,      # Width of toolbar buttons
    'button_height': None,   # Height of buttons (defaults to height - 5)
    'button_spacing': 50,    # Space between buttons

    # Positioning
    'possize': {
        'x': 0,             # X position relative to window
        'y': 0,             # Y position relative to window
        'width': None,      # Width (defaults to window width)
        'height': None      # Height (defaults to toolbar height)
    },

    # Colors (in hex format: 0xRRGGBB)
    'colors': {
        'border': 0x000000,      # Border line color
        'button_normal': 0xE8E8E8,  # Normal button state
        'button_hover': 0xF5F5F5,   # Mouse hover state
        'button_pressed': 0xC0C0C0  # Pressed state
    }
}
```

Toolbar Items

Toolbar items are defined as lists with the following format:

`[Type, Label, Icon, Callback, Tooltip, Width_offset]`

- **Type**: Either 'Button' or 'Line' (for separators)
- **Label**: Text displayed under the button
- **Icon**: Icon filename (must be in TOOLBAR_GRAPHICS_DIR)
- **Callback**: Function to call when button is clicked
- **Tooltip**: Hover text for the button
- **Width_offset**: Additional width for the button (if needed)

Window Resize Handling

To properly handle window resizing with a toolbar, implement the `window_resizing` method in your main application if you haven't already:

```
def window_resizing(self, width, height):
    """Handle window resize events"""
    if hasattr(self, 'toolbar_manager'):
        self.toolbar_manager.resize(width, height)
```

Important Notes:

- Toolbar icons must be placed in the `TOOLBAR_GRAPHICS_DIR` directory
- The toolbar automatically handles mouse events and visual feedback
- Separators ('Line' type) create vertical lines between button groups
- The toolbar will automatically resize with the window
- If `button_height` is not specified, it defaults to `height - 5` to account for the border
- Colors are specified in hexadecimal format (0xRRGGBB)
- The toolbar includes a bottom border line that is automatically positioned

Creating Dialogs in PyBrex

PyBrex provides a dialog system built on top of LibreOffice's UNO dialog framework. The system consists of three main components:

1. `DialogBase` - The base dialog class that handles dialog creation and lifecycle
2. `Controls` - A rich set of UI control creation and management functions

3. `Listeners` - Event handling system for dialog controls

Basic Dialog Structure

To create a dialog, you'll need to create a class that inherits from `DialogBase`. Here's a basic example:

```
from librepy.pybrex.dialog import DialogBase

class MyDialog(DialogBase):

    # Define dialog size (x, y, width, height)
    POS_SIZE = 0, 0, 260, 180

    # Set whether dialog should be disposed after closing
    DISPOSE = True

    def __init__(self, ctx, cast, **props):
        DialogBase.__init__(self, ctx, cast, **props)

    def _create(self):
        'Create dialog controls'
        # Add controls here
        pass

    def _prepare(self):
        'Called before dialog is shown'
        pass

    def _done(self, ret):
        'Called when dialog is closed'
        return ret
```

Adding Controls

The `Controls` class provides methods for adding LibreOffice UI elements. Here are some of the most common controls:

```
def _create(self):
    # Add OK/Cancel buttons
    self.add_ok_cancel()

    # Basic positioning
    x, y = 15, 10

    # Add a group box
    self.add_groupbox('group1', x-4, y, 240, 120,
                      Label='Group Title',
                      FontWeight=110)

    # Add a label and text field
    self.add_label('label1', x, y, 100, 11,
                   Label='Name:')
    self.add_edit('edit1', x, y+13, 150, 13)

    # Add checkboxes
    self.add_check('check1', x, y, 100, 11,
                   Label='Enable feature',
                   callback=self.on_check_clicked)

    # Add radio buttons
    self.add_radio('radio1', x, y, 100, 11,
                   Label='Option 1',
```

```
                    callback=self.on_radio_clicked)

    # Add combo box
    self.add_combo('combo1', x, y, 100, 20)

    # Add numeric field
    self.add_numeric('num1', x, y, 60, 13,
                     data_type='float',
                     DecimalAccuracy=2)
```

Data Types and Values

The dialog system includes automatic value handling through the `data_type` parameter:

- `'str'` - String values
- `'int'` - Integer values
- `'float'` - Floating point values
- `'date'` - Date values
- `'time'` - Time values
- `'check'` - Boolean checkbox values
- `'option'` - Radio button state
- `'item'` - Selected item in lists/combos

Values are automatically handled through the `get_values()` and `set_values()` methods:

```
def _prepare(self):
    # Clear existing values
    self.clear_values()

    # Set initial values
    self.set_values({
        'edit1': 'Initial text',
        'num1': 42.5,
        'check1': True
    })

def _done(self, ret):
    if ret == 1:  # OK button clicked
        # Get updated values
        values = self.get_values()
    return ret, values
```

Event Handling

The `Listeners` system provides methods to handle control events:

```
def _create(self):
    # Add button with click handler
    btn = self.add_button('btn1', x, y, 50, 15,
                          Label='Click Me',
                          callback=self.on_button_click)

    # Add text field with change handler
    edit = self.add_edit('edit1', x, y, 100, 15)
    self.add_text_listener(edit, self.on_text_changed)

    # Add list with selection handler
    lst = self.add_list('list1', x, y, 100, 60)
    self.add_item_listener(lst, self.on_selection_changed)

def on_button_click(self, event):
    # Handle button click
```

```
    pass

def on_text_changed(self, event):
    # Handle text change
    pass

def on_selection_changed(self, event):
    # Handle selection change
    pass
```

Available listener types: - `add_action_listener` - Button clicks - `add_item_listener` - List/combo selection changes - `add_text_listener` - Text field changes - `add_focus_listener` - Focus gain/loss - `add_key_listener` - Keyboard events - `add_mouse_listener` - Mouse events - `add_adjustment_listener` - Scrollbar adjustments

Complete Dialog Example

Here's a complete example of a dialog that collects company information:

```
from librepy.pybrex.dialog import DialogBase
from librepy.pybrex.dialogs.misc_dialogs import FilePickerDlg

class CompanyInfoDialog(DialogBase):

    POS_SIZE = 0, 0, 260, 180
    DISPOSE = True

    def __init__(self, ctx, cast, **props):
        super().__init__(ctx, cast, **props)

    def _create(self):
        self.add_ok_cancel()

        x, y = 15, 10

        # Add group box
        self.add_groupbox('infobox', x-4, y, 240, 120,
                          Label='Company Info',
                          FontWeight=110)

        # Company name
        y += 10
        self.add_label('NameLabel', x, y, 100, 11,
                       Label='~Name:')
        self.add_edit('name', x, y+13, 150, 13)

        # Phone numbers
        y += 33
        self.add_label('Phone1Label', x, y, 70, 11,
                       Label="Phone 1:")
        self.add_edit('phone1', x, y+13, 70, 13)

        self.add_label('Phone2Label', x+80, y, 70, 11,
                       Label="Phone 2:")
        self.add_edit('phone2', x+80, y+13, 70, 13)

        self.add_label('FaxLabel', x+160, y, 70, 11,
                       Label="Fax:")
        self.add_edit('fax', x+160, y+13, 70, 13)

        # File picker
        y += 40
```

```
        self.add_label('pathLocLabel', x, y, 50, 11,
                       Label='Path:')
        y += 13
        self.add_edit('path_location', x, y, 200, 11)
        self.add_button('browse_path_loc', x+210, y, 15, 13,
                        Label='...',
                        callback=self.browse_path_clicked)

    def browse_path_clicked(self, event):
        ctr = self._controls['path_location']
        f = self.select_file(ctr.Text)
        if f:
            ctr.Text = uno.fileUrlToSystemPath(f[0])

    def select_file(self, base_path,
                    filters=(('Image files', '*.png'),
                             ('All files', '*'))):
        dlg = FilePickerDlg(self.ctx, self.cast)
        return dlg.execute(base_path, filters=filters)

    def _prepare(self):
        self.clear_values()
        self.set_values(self._values)
        self._controls['name'].setFocus()

    def _done(self, ret):
        if ret == 1:  # OK clicked
            self.get_values()
        return ret, self._values

    def execute(self, values):
        self._values = values.copy()
        return DialogBase.execute(self)
```

To use this dialog:

```
def new_contact(self, *args):
    # Create dialog with initial values
    dlg = CompanyInfoDialog(self.ctx, self.smgr)
    values = {
        'name': 'John Doe',
        'phone1': '999-999-9999',
        'phone2': '999-999-9998',
        'fax': '999-999-9997',
        'path_location': '/path/to/file'
    }

    # Show dialog and get results
    ret, values = dlg.execute(values)

    if ret == 1:  # OK clicked
        # Use updated values
        print(values)
```

Creating a Contact Dialog

To continue with the contact list application, let's create a file named contact_dialog.py and add the following code to it.

```
import uno

from librepy.pybrex.dialog import DialogBase
```

```
from librepy.pybrex.dialogs.misc_dialogs import FilePickerDlg

class ContactDialog(DialogBase):

    POS_SIZE = 0, 0, 260, 180
    DISPOSE = True

    def __init__(self, ctx, cast, **props):
        super().__init__(ctx, cast, **props)

    def _create(self):
        self.add_ok_cancel()

        x, y = 15, 10

        # Add group box
        self.add_groupbox('infobox', x-4, y, 240, 120,
                          Label='Company Info',
                          FontWeight=110)

        # Company name
        y += 10
        self.add_label('NameLabel', x, y, 100, 11,
                       Label='~Name:')
        self.add_edit('name', x, y+13, 150, 13)

        # Phone numbers
        y += 33
        self.add_label('Phone1Label', x, y, 70, 11,
                       Label="Phone 1:")
        self.add_edit('phone1', x, y+13, 70, 13)

        self.add_label('Phone2Label', x+80, y, 70, 11,
                       Label="Phone 2:")
        self.add_edit('phone2', x+80, y+13, 70, 13)

        self.add_label('FaxLabel', x+160, y, 70, 11,
                       Label="Fax:")
        self.add_edit('fax', x+160, y+13, 70, 13)

        # File picker
        y += 40
        self.add_label('pathLocLabel', x, y, 50, 11,
                       Label='Path:')
        y += 13
        self.add_edit('path_location', x, y, 200, 11)
        self.add_button('browse_path_loc', x+210, y, 15, 13,
                        Label='...', callback=self.browse_path_clicked)
        self.add_check("is_active", x, y+30, 150, 13, Label="Active")

    def browse_path_clicked(self, event):
        ctr = self._controls['path_location']
        f = self.select_file(ctr.Text)
        if f:
            ctr.Text = uno.fileUrlToSystemPath(f[0])

    def select_file(self, base_path,
                    filters=(('Image files', '*.png'),
                             ('All files', '*'))):
        dlg = FilePickerDlg(self.ctx, self.cast)
```

```
        return dlg.execute(base_path, filters=filters)

    def _prepare(self):
        self.clear_values()
        self.set_values(self._values)
        self._controls['name'].setFocus()

    def _done(self, ret):
        if ret == 1:  # OK clicked
            self.get_values()
        return ret, self._values

    def execute(self, values):
        self._values = values.copy()
        return DialogBase.execute(self)
```

This covers the basics of dialog creation in PyBrex. For more advanced features, check out the dialog.py, controls.py and listeners.py files in the PyBrex source code.

Executing the Dialog

We will now link this dialog to a toolbar button. To execute the dialog, we need to add the following code to the new_contact method in the toolbar_manager.py file.

```
def new_contact(self, *args):
    """Create new contact"""
    print("New contact")
    try:
        dialog = ContactDialog(self.ctx, self.smgr)
        values = {
            'name': 'John Doe',
            'phone1': '999-999-9999',
            'phone2': '999-999-9998',
            'fax': '999-999-9997',
            'path_location': '/path/to/file'
        }
        result, values = dialog.execute(values)

        if result == 1:  # OK was clicked
            self.logger.info("New contact created")
            print(values)
            # TODO: Save the contact data
            self.logger.debug(f"Contact values: {values}")
        else:
            print("New contact cancelled")

    except Exception as e:
        print(f"Error creating new contact: {e}")
```

Don't forget to import the the contact_dialog.py file on the top of the toolbar_manager.py file.

```
from librepy.contact_list.contact_dialog import ContactDialog
```

This will create a new contact dialog and execute it when the new contact button is clicked.

Adding Controls to Frames

PyBrex provides a flexible system for adding controls (buttons, labels, etc.) to your application's main window. This is done through a container system that manages the controls' lifecycle and positioning.

Creating a Container

Before adding any controls, you first need to create a container that will hold them:

```
from librepy.pybrex import ctr_container

class MyControlManager:
    def __init__(self, parent, ctx, smgr, frame):
        self.parent = parent
        self.ctx = ctx
        self.smgr = smgr
        self.frame = frame
        self.logger = parent.logger

        # Define the container position and size (x, y, width, height)
        self.ps = (20, 40, 600, 400)

        # Create the main container
        self.container = ctr_container.Container(
            ctx,
            smgr,
            frame.window,
            self.ps,
            background_color=0xDCDAD5  # Light gray background
        )
```

Adding Basic Controls

Once you have a container, you can add various controls to it. The container provides methods similar to dialog controls:

```
def create_controls(self):
    x, y = 20, 20  # Starting position

    # Add a button
    self.reload_btn = self.container.add_button(
        'reload_btn',      # Control name
        x, y,              # Position
        100, 20,           # Width, Height
        Label='Reload'     # Button text
    )

    # Add a label
    y += 30
    self.status_label = self.container.add_label(
        'status_label',
        x, y,
        200, 20,
        Label='Ready'
    )

    # Add an edit field
    y += 30
    self.search_field = self.container.add_edit(
        'search_field',
        x, y,
        150, 20,
        Text='Search...'
    )
```

Event Handling

The container system supports the same event listeners as dialogs:

```
def setup_listeners(self):
    # Button click handler
    self.container.add_action_listener(
        self.reload_btn,
        self.on_reload_clicked
    )

    # Text change handler
    self.container.add_text_listener(
        self.search_field,
        self.on_search_changed
    )

def on_reload_clicked(self, event):
    self.logger.info("Reload button clicked")
    # Handle reload action

def on_search_changed(self, event):
    search_text = event.Source.Text
    self.logger.info(f"Search text changed: {search_text}")
    # Handle search text change
```

Resource Management

Always implement proper cleanup in your manager class:

```
def dispose(self):
    """Clean up resources"""
    try:
        self.container.dispose()
    except Exception as e:
        self.logger.error(f"Error disposing container: {e}")
        self.logger.error(traceback.format_exc())
```

Important Notes:

- The container uses the same coordinate system as dialogs (0,0 is top-left)
- Controls are automatically disposed when the container is disposed
- The container background color can be customized during initialization
- Control names must be unique within the container
- The container will automatically handle window resize events
- All controls support the same properties as their dialog counterparts

Common Control Types

- `add_button()` - Push buttons
- `add_label()` - Text labels
- `add_edit()` - Text input fields
- `add_check()` - Checkboxes
- `add_combo()` - Dropdown lists
- `add_numeric()` - Number input fields
- `add_groupbox()` - Visual grouping of controls
- `add_grid()` - Data grids (covered in detail in the Grid Controls section)

For more a comprehensive list of controls please refer to the controls.py file in the PyBrex source code.

Grid Controls

Grid controls allow you to display tabular data in both dialogs and containers. They provide a way to show and interact with structured data in your application. There are two ways to add a grid depending on where you want to use it:

- For dialogs: Use the `add_grid()` method inherited from the dialog class (defined in controls.py)
- For containers: Use the `add_grid()` method from the container class (defined in ctr_container.py)

After creating a grid, you can interact with it using the various methods provided by the grid.py module, which we'll cover in detail below.

Creating a Grid

There are two ways to create a grid, depending on whether you're using a dialog or a container:

1. **In Dialogs:**

```
# Inside your dialog's _create method
titles = [
    ("Name", "name", 100, 1),      # (Header, Data Key, Width, Type)
    ("Active", "active", 50, 2),   # Type 2 = Boolean (Yes/No)
    ("Notes", "notes", 150, 3),    # Type 3 = Text with newlines
]

grid_base, grid_control = self.add_grid(
    'my_grid',      # Control name
    x=10, y=10,     # Position
    width=300,      # Width
    height=200,     # Height
    titles=titles,  # Column definitions
    # Optional properties:
    SelectionModel=1,      # Single row selection
    UseGridLines=True,     # Show grid lines
    ShowColumnHeader=True  # Show column headers
)
```

2. **In Containers:**

```
# Inside your container class
titles = [
    ("Name", "name", 100, 1),
    ("Active", "active", 50, 2),
    ("Notes", "notes", 150, 3)
]

grid_base, grid_control = self.add_grid(
    'my_grid',
    x=10, y=10,
    width=300, height=200,
    titles=titles
)
```

In both cases, `add_grid()` returns a tuple containing: - `grid_base`: A GridBase instance that provides methods for manipulating the grid data (clear, add, update, etc.) - `grid_control`: The actual UNO grid control object that represents the visual component

You'll primarily use `grid_base` for interacting with the grid's data and functionality, while `grid_control` is used internally and for advanced UNO-specific operations.

Column Types

The `titles` parameter defines your grid columns as a list of tuples. Each tuple contains: - Column header text - Data field key - Column width in pixels - Column type: - 1: Regular text - 2: Boolean (displays as "Y"/"N") - 3: Text with newlines (displayed as comma-separated)

Setting Grid Data

The grid accepts data as a list of dictionaries, where each dictionary represents a row:

```
data = [
    {
        'id': 1,              # Required for row identification
        'name': 'John Doe',
        'active': True,
        'notes': 'Line 1\nLine 2'
    },
    {
        'id': 2,
        'name': 'Jane Smith',
        'active': False,
        'notes': 'Some notes'
    }
]

# Set data with default options
grid_base.set_data(data)

# Set data with custom options
grid_base.set_data(
    data,
    heading='id',   # Field to use as row identifier
    clear=True,     # Clear existing data first
    resort=True     # Maintain current sort order
)
```

Grid Methods

The GridBase class provides several methods for interacting with the grid:

```
# Clear all rows
grid_base.clear()

# Get the currently selected row's heading (id)
row_id = grid_base.active_row_heading()

# Reload grid with new data
grid_base.reload(new_data)

# Update a specific row
grid_base.update(row_data)  # Must have matching 'id'

# Delete the selected row
grid_base.delete()

# Add a new row
grid_base.add(row_data, heading='new_id')

# Change the background color of the last row
grid_base.set_last_line_color(0xFF0000)  # Red
```

Row Colors

By default, grids use alternating row colors for better readability. You can customize these colors when creating the grid:

```
grid_base, grid_control = self.add_grid(
    'my_grid', x, y, width, height, titles,
    color1=0xE8E8E8,  # Light gray
```

```
    color2=0xFFFFFF   # White
)
```

Double-Click Handling

You can add a double-click handler to the grid:

```
def on_row_double_click(self, event):
    row_id = self.grid_base.active_row_heading()
    if row_id:
        print(f"Double clicked row {row_id}")

# Set the handler
grid_base.mouse_doubleclick_fn = self.on_row_double_click
```

Grid Properties

Common grid properties you can set: - `SelectionModel`: Row selection mode (1 = single row) - `UseGridLines`: Show/hide grid lines - `ShowColumnHeader`: Show/hide column headers - `ShowRowHeader`: Show/hide row headers - `HScroll`: Enable horizontal scrolling - `VScroll`: Enable vertical scrolling - `HeaderBackgroundColor`: Color of header row - `RowBackgroundColors`: Tuple of alternating row colors

Example with properties:

```
grid_base, grid_control = self.add_grid(
    'my_grid', x, y, width, height, titles,
    SelectionModel=1,
    UseGridLines=True,
    ShowColumnHeader=True,
    ShowRowHeader=False,
    HScroll=True,
    VScroll=True,
    HeaderBackgroundColor=0xE0E0E0
)
```

Important Notes:

- Always include an ‘id’ field in your data for row identification
- Column types affect how data is displayed:
  - Type 2 (Boolean) converts True/False to “Y”/“N”
  - Type 3 (Multiline) converts newlines to commas
- The grid automatically handles:
  - Column sorting
  - Row selection
  - Scrolling
  - Alternating row colors
- Row operations (update/delete) work on the currently selected row
- The grid maintains sort order when reloading data (if resort=True)