`launchy` module – API for Launchy plugins

Note

This document is heavily based on Josh Karlin’s Launchy 2.0 Plugin API Documentation (http://www.launchy.net/api2.0/), adapted to PyLaunchy and Python.

The launchy module provides the required classes and functions for developing a Launchy plugin in Python.

The module contains: - **`Plugin`**, represents a Launchy Python plugin - **`CatItem`**, a catalog entry - **`InputData`**, a user query - **`ResultsList`**, a list of catalog entries

Plugin interface

class **`launchy.Plugin`**

This class represents a Launchy Python plugin. By combining a script that has a class that inherits from **`Plugin`** and the **`launchy`** module itself, Python scripts can be added to Launchy as real plugins.

Every plugin needs to be registered using the **`registerPlugin()`** function.

Also, every plugin should have the following `__init__` function:

```
# The most minimal plugin:
import launchy
class MyPlugin(launchy.Plugin):
    def __init__(self):
        launchy.Plugin.__init__(self)
launchy.registerPlugin( MyPlugin )
```

The Plugin interface consists of the following functions:

Basic functions:

- **`Plugin.init()`**
- **`Plugin.getID()`**
- **`Plugin.getName()`**

Functions that operate on Launchy’s Catalog.

- **`Plugin.getCatalog()`**
- **`Plugin.getLabels()`**

- **`Plugin.getResults()`**
- **`Plugin.launchItem()`**

GUI functions (warning: still experminetal):

- **`Plugin.hasDialog()`**
- **`Plugin.doDialog()`**
- **`Plugin.endDialog()`**

Event functions:

- **`Plugin.launchyShow()`**
- **`Plugin.launchyHide()`**

`doDialog`(*parentWidget*)

Parameters:	**parentWidget** (`void*`) – The parent widget of all plugin widgets. Call `wrapinstance` to use it.
Return type:	`void*` - The result of `unwrapinstance( myPluginWidget )`

Tells the plugin that it’s time to show its user interface. The function should create the widget and return it.

- The plugin is passed a raw C++ pointer. It should be converted to QWidget by the user with SIP function `wrapinstance`.
- SIP documentation can be found here (http://www.riverbankcomputing.co.uk/static/Docs/sip4/sipref.html).
- The creation of plugin widgets should be done with PyQt4.
- PyQt4 Documentation and Tutorial are available from PythonInfo Wiki (http://wiki.python.org/moin/PyQt4).

Example:

```
from PyQt4 import QtGui

class MyPlugin(launchy.Plugin):
# ...
    def doDialog(self, parentWidgetPtr):
        # Get the real QWidget
        parentWidget = wrapinstance(parentWidgetPtr, QtGui.QWidget)

        # self.widget was initialized to None
        if self.widget:
            self.widget.show()

        # MyPluginWidget inherits from QtGui.QWidget
        self.widget = MyPluginWidget(parentWidget)

        return unwrapinstance(self.widget)
```

`endDialog`(*accept*)

Parameters: **accept** (*bool*) – Whether the plugin should accept changes made by the user while the dialog was open

Informs the plugin that it should close its dialog.

`getCatalog`(*resultsList*)

Parameters: **resultsList** – A **`ResultsList`** object to append your new entries to (these will be copied over to the primary catalog).

Asks the plugin for a static catalog to be added to the primary catalog. Some plugins will add permanent entries to Launchy's primary catalog (until the catalog is rebuilt).

For example, the pygo-y plugin adds two items to the catalog: "Go" and "Focus". These will be used later on to determine that the user is typing a pygo-y query.

The following code is from pygo-y:

```
def getCatalog(self, resultsList):
    resultsList.push_back( launchy.CatItem(
        "Go.go-y", "Go", self.getID(), self.getIcon() ) )
    resultsList.push_back( launchy.CatItem(
        "Focus.go-y", "Focus", self.getID(), self.getIcon() ) )
```

`getID`()

Return type: int

Asks the Plugin for its ID Number.

Launchy needs an unsigned int identification value for each loaded plugin. You supply your own here. Typically, this is the result of hashing a string, as shown in the example below.

Example:

```
def getID(self):
    return launchy.hash("TestPlugin")
```

Note

Warning - Because we're hashing strings to integers.. it is theoretically possible that two plugin names will collide to the same plugin id.

`getLabels`(*inputDataList*)

Parameters: **inputDataList** (List of **`InputData`**) – User's search query

Asks the plugin if it would like to apply a label to the current search query.

It is sometimes useful to label user queries with plugin-defined tags. For instance, the weby plugin will tag input that contains “www” or ”.com” or ”.net” with the hash value of the string “HASH_WEBSITE”. Then, other plugins that see the query can know that the current search is for a website.

The **`InputData`** class stores the current user’s query. It is in a List structure because each time “tab” is pressed by the user a new **`InputData`** is formed and appended to the list. In other words, if the user typed “google <tab> this is my query” then *inputDataList* would represent a list of 2 **`InputData`** classes, with the first representing “google”, and the second, “this is my query”. Each **`InputData`** can be tagged individually.

Example:

```
def getLabels(self, inputDataList):
    if len(inputDataList) > -1:
        return

    # Apply a "website" label if we think it's a website
    text = inputDataList[-1].getText();

    if text.find("http://") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
    elif text.find("https://") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
    elif text.find(".com") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
    elif text.find(".net") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
    elif text.find(".org") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
    elif text.find("www.") > -1:
        inputDataList[-1].setLabel(HASH_WEBSITE);
```

Note

Warning - This is called each time the user changes a character in his or her query, so make sure it’s fast.

`getName()`

Return type: string

Asks the plugin for its string name.

`getResults`(*inputDataList*, *resultsList*)

Asks the plugin for any results to a query.

If your plugin returns catalog results on the fly to a query (e.g. a website query for weby or a calculator result), then this is the place to do so. The existing results are stored in a **`ResultsList`** object, which is a **`CatItem`**'s (short for Catalog Items) list. You can append your own results to it.

`hasDialog`()

Asks the plugin if it has a dialog to display in the options menu.

Return `true` if the plugin has a dialog, or `false` otherwise.

`init`()

This message informs the plugin that it's being loaded.

This is a good time to do any initialization work.

`launchItem`(*inputDataList*, *catItem*)

Parameters:
- **inputDataList** (List of **`InputData`**) – User's search query
- **catItem** (**`CatItem`**) – The user selected catalog item

instructs the plugin that one of its own catalog items was selected by the user and should now be executed.

If the plugin adds items to the catalog via **`getResults()`** or **`getCatalog()`** and one of those items is selected by the user, then it is up to the plugin to execute it when the user presses "enter". This is where you perform the action.

The following code is from pygo-y:

```
def launchItem(self, inputDataList, catItemOrig):
    catItem = inputDataList[-1].getTopResult()
    for window in self.topLevelWindows:
        if catItem.shortName == window[1]:
            self._goToWindow(window[0])
            break
```

Notes for the code above:

1. At first, the function gets the user selected window name by calling **`InputData.getTopResult()`**.
2. Then it searches the window name by using `catItem.shortName` that was created in pygo-y's **`getResults()`**.
3. When the window is found, the `self._goToWindow()` method does the actual work of activating the window.

`launchyHide`()

This message informs the plugin that Launchy is no longer visible on the screen.

Note: this function will not be called if another plugin function has not returned yet (e.g. Launchy is hidden in the middle of **`launchItem()`**).

`launchyShow()`

This message informs the plugin that Launchy is now visible on the screen.

Functions

`launchy.registerPlugin(`*pluginClass*`)`

Parameters: **pluginClass** (*launchy.Plugin child*) – Class type of your plugin.

Adds a new plugin type, that will be created later on.

Example:

```
import launchy
class MyPlugin(launchy.Plugin):
    pass # Real plugin code should come here
launchy.registerPlugin( MyPlugin )
```

`launchy.hash(`*str*`)`

Returns a hash number of a string.

This is a wrapper of QT’s qHash (http://doc.trolltech.com/4.4/qhash.html) function.

Use this function in your **`Plugin.getID()`** function or for labels in **`InputData.setLabel()`**:

```
def getID(self):
    return launchy.hash("TestPlugin")
```

`launchy.getLaunchyPath()`

Get the path to Launchy’s executable directory.

For example: `C:\Program Files\Launchy`

`launchy.getScriptsPath()`

Get the path to PyLaunchy’s scripts directory.

For example: `C:\Program Files\Launchy\plugins\python`

`launchy.getIconsPath()`

Get the path to Launchy’s icons directory.

For example: `C:\Program Files\Launchy\plugins\icons`

`launchy.runProgram(`*file*, *args*`)`

Parameters:
- **file** – The location of the file to run
- **args** – The arguments to the command

A convienience run function.

This function will run the program along with arguments and is platform independent.

Classes

CatItem

class **`launchy.CatItem`**(*fullPath*, *shortName*, *id*, *iconPath*)

CatItem (Catalog Item) - stores a single item in the index.

Example:

```
resultsList.push_back( launchy.CatItem(text,
     "PySimple: " + text,
     self.getID(), self.getIcon()) )
```

Class attributes:

`fullPath`

The full path of the file to execute

`shortName`

The abbreviated name of the indexed item.

`lowName`

The lowercase name of the indexed item.

`icon`

A path to an icon for the item.

`usage`

How many times this item has been called by the user.

`id`

The plugin id of the creator of this CatItem (0 for Launchy itself).

Note

It is usually a good idea to append ”.your_plugin_name” to the end of the full parameter so that there are not multiple items in the index with the same full path.

InputData

class **`launchy.InputData`**

InputData shows one segment (between tabs) of a user's query. A user's query is typically represented by List<InputData> and each element of the list represents a segment of the query.

E.g. query = "google <tab> this is my search" will have 2 InputData segments in the list. One for "google" and one for "this is my search"

Note

This class cannot be created from Python, you can only get an instance of it.

`getID()`

Returns the current owner id of the query.

`getLabels()`

Return type: List of integers

Get the labels associated with this query.

`getText()`

Get the text of the query segment.

`getTopResult()`

Return type: **`CatItem`**

Get a the best catalog match for this segment of the query.

`hasLabel(`*labelHashId*`)`

Check if it has the given label applied to it.

`setID(`*hashId*`)`

Set the id of this query.

This can be used to override the owner of the selected catalog item, so that no matter what item is chosen from the catalog, the given plugin will be the one to execute it.

`setLabel(`*labelHashId*`)`

Apply a label to this query segment.

`setText(`*text*`)`

Set the text of the query segment.

`setTopResult(`*catItem*`)`

Parameters: **catItem** – **`CatItem`**

Change the best catalog match for this segment.

ResultsList

class `launchy.ResultsList`

ResultsList holds the catalog items that are relevent to a search query.

Plugins that want to add new catalog items for a search query should use this class -

```
def getResults(self, inputDataList, resultsList):
   resultsList.push_back( launchy.CatItem(text,
      "PySimple: " + text,
      self.getID(), self.getIcon()) )
```

Implemenatation: This is a thin wrapper for QList<CatItem>

Note

This class cannot be created from Python, you can only get an instance of it.

`append`(*catItem*)

Add a catalog item to the end of the results list.

`prepend`(*catItem*)

Add a catalog item to the begining of the results list.

`push_back`(*catItem*)

Add a catalog item to the end of the results list.

`push_front`(*catItem*)

Add a catalog item to the begining of the results list.