rumps Documentation

Release 0.2.0

Jared Suttles

June 24, 2015

Contents

rumps is...

Ridiculously Uncomplicated Mac os x Python Statusbar apps!

rumps exposes Objective-C classes as Python classes and functions which greatly simplifies the process of creating a statusbar application.

Say you have a Python program and want to create a relatively simple interface for end user interaction on a Mac. There are a number of GUI tools available to Python programmers (PyQt, Tkinter, PyGTK, WxPython, etc.) but most are overkill if you just want to expose a few configuration options or an execution switch.

If all you want is a statusbar app, rumps makes it easy.

GitHub project: https://github.com/jaredks/rumps

Contents:

CHAPTER 1

Examples

Sometimes the best way to learn something is by example. Form your own application based on some of these samples.

1.1 Simple subclass structure

Just a straightforward application,

```
import rumps

class AwesomeStatusBarApp(rumps.App):
    def __init__(self):
        super(AwesomeStatusBarApp, self).__init__("Awesome App")
        self.menu = ["Preferences", "Silly button", "Say hi"]

    @rumps.clicked("Preferences")
    def prefs(self, _):
        rumps.alert("jk! no preferences available!")

    @rumps.clicked("Silly button")
    def onoff(self, sender):
        sender.state = not sender.state

    @rumps.clicked("Say hi")
    def sayhi(self, _):
        rumps.notification("Awesome title", "amazing subtitle", "hi!!1")

if __name__ == "__main__":
    AwesomeStatusBarApp().run()
```

1.2 Decorating any functions

The following code demonstrates how you can decorate functions with *rumps.clicked()* whether or not they are inside a subclass of *rumps.App*. The parameter `sender`, the *rumps.MenuItem* object, is correctly passed to both functions even though `button` needs an instance of `SomeApp` as its `self` parameter.

Usually functions registered as callbacks should accept one and only one argument but an *App* subclass is viewed as a special case as its use can provide a simple and pythonic way to implement the logic behind an application.

```
from rumps import *

@clicked('Testing')
def tester(sender):
    sender.state = not sender.state

class SomeApp(rumps.App):
    def __init__(self):
        super(SomeApp, self).__init__(type(self).__name__, menu=['On', 'Testing'])
        rumps.debug_mode(True)

    @clicked('On')
    def button(self, sender):
        sender.title = 'Off' if sender.title == 'On' else 'On'
        Window("I can't think of a good example app...").run()

if __name__ == "__main__":
    SomeApp().run()
```

1.3 New features in 0.2.0

Menu items can be disabled (greyed out) by passing `None` to *rumps.MenuItem.set_callback()*. *rumps.alert()* no longer requires *title* (will use a default localized string) and allows for custom *cancel* button text. The new parameter *quit_button* for *rumps.App* allows for custom quit button text or removal of the quit button entirely by passing `None`.

> **Warning:** By setting *rumps.App.quit_button* to `None` you **must include another way to quit the application** by somehow calling *rumps.quit_application()* otherwise you will have to force quit.

```
import rumps

rumps.debug_mode(True)

@rumps.clicked('Print Something')
def print_something(_):
    rumps.alert(message='something', ok='YES!', cancel='NO!')


@rumps.clicked('On/Off Test')
def on_off_test(_):
    print_button = app.menu['Print Something']
    if print_button.callback is None:
        print_button.set_callback(print_something)
    else:
        print_button.set_callback(None)


@rumps.clicked('Clean Quit')
def clean_up_before_quit(_):
    print 'execute clean up code'
    rumps.quit_application()


app = rumps.App('Hallo Thar', menu=['Print Something', 'On/Off Test', 'Clean Quit'], quit_button=None
app.run()
```

CHAPTER 2

Creating Standalone Applications

If you want to create your own bundled .app you need to download py2app: https://pythonhosted.org/py2app/

For creating standalone apps, just make sure to include `rumps` in the `packages` list. Most simple statusbar-based apps are just "background" apps (no icon in the dock; inability to tab to the application) so it is likely that you would want to set `'LSUIElement'` to `True`. A basic `setup.py` would look like,

```
from setuptools import setup

APP = ['example_class.py']
DATA_FILES = []
OPTIONS = {
    'argv_emulation': True,
    'plist': {
        'LSUIElement': True,
    },
    'packages': ['rumps'],
}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

With this you can then create a standalone,

```
python setup.py py2app
```

CHAPTER 3

Debugging Your Application

When writing your application you will want to turn on debugging mode.

```
import rumps
rumps.debug_mode(True)
```

If you are running your program from the interpreter, you should see the informational messages.

```
python {your app name}.py
```

If testing the .app generated using py2app, to be able to see these messages you must not,

```
open {your app name}.app
```

but instead run the executable. While within the directory containing the .app,

```
./{your app name}.app/Contents/MacOS/{your app name}
```

And, by default, your .app will be in `dist` folder after running `python setup.py py2app`. So of course that would then be,

```
./dist/{your app name}.app/Contents/MacOS/{your app name}
```

CHAPTER 4

rumps Classes

4.1 App

class rumps.**App**(*name*, *title=None*, *icon=None*, *menu=None*, *quit_button='Quit'*)

Represents the statusbar application.

Provides a simple and pythonic interface for all those long and ugly *PyObjC* calls. *rumps.App* may be subclassed so that the application logic can be encapsulated within a class. Alternatively, an *App* can be instantiated and the various callback functions can exist at module level.

Changed in version 0.2.0: *name* parameter must be a string and *title* must be either a string or None. *quit_button* parameter added.

Parameters

- **name** – the name of the application.
- **title** – text that will be displayed for the application in the statusbar.
- **icon** – file path to the icon that will be displayed for the application in the statusbar.
- **menu** – an iterable of Python objects or pairs of objects that will be converted into the main menu for the application. Parsing is implemented by calling *rumps.MenuItem.update()*.
- **quit_button** – the quit application menu item within the main menu. If None, the default quit button will not be added.

icon

A path to an image representing the icon that will be displayed for the application in the statusbar. Can be None in which case the text from title will be used.

Changed in version 0.2.0: If the icon is set to an image then changed to None, it will correctly be removed.

menu

Represents the main menu of the statusbar application. Setting *menu* works by calling *rumps.MenuItem.update()*.

name

The name of the application. Determines the application support folder name. Will also serve as the title text of the application if title is not set.

open(**args*)

Open a file within the application support folder for this application.

```
app = App('Cool App')
with app.open('data.json') as f:
    pass
```

Is a shortcut for,

```
app = App('Cool App')
filename = os.path.join(application_support(app.name), 'data.json')
with open(filename) as f:
    pass
```

quit_button
The quit application menu item within the main menu. This is a special *rumps.MenuItem* object that will both replace any function callback with *rumps.quit_application()* and add itself to the end of the main menu when *rumps.App.run()* is called. If set to None, the default quit button will not be added.

> **Warning:** If set to None, some other menu item should call *rumps.quit_application()* so that the application can exit gracefully.

New in version 0.2.0.

run(***options*)
Performs various setup tasks including creating the underlying Objective-C application, starting the timers, and registering callback functions for click events. Then starts the application run loop.

Changed in version 0.2.1: Accepts *debug* keyword argument.

Parameters **debug** – determines if application should log information useful for debugging. Same effect as calling *rumps.debug_mode()*.

title
The text that will be displayed for the application in the statusbar. Can be None in which case the icon will be used or, if there is no icon set the application text will fallback on the application name.

Changed in version 0.2.0: If the title is set then changed to None, it will correctly be removed. Must be either a string or None.

4.2 MenuItem

class rumps.**MenuItem**(*title*, *callback=None*, *key=None*, *icon=None*, *dimensions=None*)
Represents an item within the application's menu.

A *rumps.MenuItem* is a button inside a menu but it can also serve as a menu itself whose elements are other *MenuItem* instances.

Encapsulates and abstracts Objective-C NSMenuItem (and possibly a corresponding NSMenu as a submenu).

A couple of important notes:

- A new *MenuItem* instance can be created from any object with a string representation.
- Attempting to create a *MenuItem* by passing an existing *MenuItem* instance as the first parameter will not result in a new instance but will instead return the existing instance.

Remembers the order of items added to menu and has constant time lookup. Can insert new *MenuItem* object before or after other specified ones.

Note: When adding a *MenuItem* instance to a menu, the value of title at that time will serve as its key for

lookup performed on menus even if the *title* changes during program execution.

Parameters

- **title** – the name of this menu item. If not a string, will use the string representation of the object.
- **callback** – the function serving as callback for when a click event occurs on this menu item.
- **key** – the key shortcut to click this menu item. Must be a string or `None`.
- **icon** – a path to an image. If set to `None`, the current image (if any) is removed.
- **dimensions** – a sequence of numbers whose length is two, specifying the dimensions of the icon.

d[key]
Return the item of d with key *key*. Raises a `KeyError` if key is not in the map.

d[key] = value
Set *d[key]* to *value* if *key* does not exist in d. *value* will be converted to a *MenuItem* object if not one already.

del d[key]
Remove *d[key]* from d. Raises a `KeyError` if *key* is not in the map.

add(*menuitem*)
Adds the object to the menu as a *rumps.MenuItem* using the *rumps.MenuItem.title* as the key. *menuitem* will be converted to a *MenuItem* object if not one already.

callback
Return the current callback function.

New in version 0.2.0.

clear()
Remove all *MenuItem* objects from within the menu of this *MenuItem*.

get($k[, d]$) $\rightarrow$ D[k] if k in D, else d. d defaults to None.

has_key(k) $\rightarrow$ True if D has a key k, else False

icon
The path to an image displayed next to the text for this menu item. If set to `None`, the current image (if any) is removed.

Changed in version 0.2.0: Setting icon to `None` after setting it to an image will correctly remove the icon. Returns the path to an image rather than exposing a *PyObjC* class.

insert_after(*existing_key*, *menuitem*)
Insert a *rumps.MenuItem* in the menu after the *existing_key*.

Parameters

- **existing_key** – a string key for an existing *MenuItem* value.
- **menuitem** – an object to be added. It will be converted to a *MenuItem* if not one already.

insert_before(*existing_key*, *menuitem*)
Insert a *rumps.MenuItem* in the menu before the *existing_key*.

Parameters

- **existing_key** – a string key for an existing *MenuItem* value.

- **menuitem** – an object to be added. It will be converted to a *MenuItem* if not one already.

items() → list of (key, value) pairs in od

iteritems()
od.iteritems -> an iterator over the (key, value) pairs in od

iterkeys() → an iterator over the keys in od

itervalues()
od.itervalues -> an iterator over the values in od

key
The key shortcut to click this menu item.

New in version 0.2.0.

keys() → list of keys in od

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

set_callback(*callback*, *key=None*)
Set the function serving as callback for when a click event occurs on this menu item. When *callback* is `None`, it will disable the callback function and grey out the menu item. If *key* is a string, set as the key shortcut. If it is `None`, no adjustment will be made to the current key shortcut.

Changed in version 0.2.0: Allowed passing `None` as both *callback* and *key*. Additionally, passing a *key* that is neither a string nor `None` will result in a standard `TypeError` rather than various, uninformative *PyObjC* internal errors depending on the object.

Parameters

- **callback** – the function to be called when the user clicks on this menu item.
- **key** – the key shortcut to click this menu item.

set_icon(*icon_path*, *dimensions=None*)
Sets the icon displayed next to the text for this menu item. If set to `None`, the current image (if any) is removed. Can optionally supply *dimensions*.

Changed in version 0.2.0: Setting *icon* to `None` after setting it to an image will correctly remove the icon. Passing *dimensions* a sequence whose length is not two will no longer silently error.

Parameters

- **icon_path** – a file path to an image.
- **dimensions** – a sequence of numbers whose length is two.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

state
The state of the menu item. The "on" state is symbolized by a check mark. The "mixed" state is symbolized by a dash.

Table 4.1: Setting states

State	Number
ON	1
OFF	0
MIXED	-1

title
The text displayed in a menu for this menu item. If not a string, will use the string representation of the object.

update(*iterable*, ***kwargs*)
Update with objects from *iterable* after each is converted to a *rumps.MenuItem*, ignoring existing keys. This update is a bit different from the usual `dict.update` method. It works recursively and will parse a variety of Python containers and objects, creating *MenuItem* object and submenus as necessary.

If the *iterable* is an instance of *rumps.MenuItem*, then add to the menu.

Otherwise, for each element in the *iterable*,

- if the element is a string or is not an iterable itself, it will be converted to a *rumps.MenuItem* and the key will be its string representation.
- if the element is a *rumps.MenuItem* already, it will remain the same and the key will be its *rumps.MenuItem.title* attribute.
- if the element is an iterable having a length of 2, the first value will be converted to a *rumps.MenuItem* and the second will act as the submenu for that *MenuItem*
- if the element is an iterable having a length of anything other than 2, a `ValueError` will be raised
- if the element is a mapping, each key-value pair will act as an iterable having a length of 2

values() → list of values in od

viewitems() → a set-like object providing a view on od's items

viewkeys() → a set-like object providing a view on od's keys

viewvalues() → an object providing a view on od's values

4.3 Window

class rumps.**Window**(*message=''*, *title=''*, *default_text=''*, *ok=None*, *cancel=None*, *dimensions=(320, 160)*)
Generate a window to consume user input in the form of both text and button clicked.

Changed in version 0.2.0: Providing a *cancel* string will set the button text rather than only using text "Cancel". *message* is no longer a required parameter.

Parameters

- **message** – the text positioned below the *title* in smaller font. If not a string, will use the string representation of the object.
- **title** – the text positioned at the top of the window in larger font. If not a string, will use the string representation of the object.
- **default_text** – the text within the editable textbox. If not a string, will use the string representation of the object.

- **ok** – the text for the "ok" button. Must be either a string or `None`. If `None`, a default localized button title will be used.
- **cancel** – the text for the "cancel" button. If a string, the button will have that text. If *cancel* evaluates to `True`, will create a button with text "Cancel". Otherwise, this button will not be created.
- **dimensions** – the size of the editable textbox. Must be sequence with a length of 2.

add_button(*name*)
Create a new button.

Changed in version 0.2.0: The *name* parameter is required to be a string.

Parameters **name** – the text for a new button. Must be a string.

add_buttons(*iterable=None*, **args*)
Create multiple new buttons.

Changed in version 0.2.0: Since each element is passed to *rumps.Window.add_button()*, they must be strings.

default_text
The text within the editable textbox. An example would be

"Type your message here."

If not a string, will use the string representation of the object.

icon
The path to an image displayed for this window. If set to `None`, will default to the icon for the application using *rumps.App.icon*.

Changed in version 0.2.0: If the icon is set to an image then changed to `None`, it will correctly be changed to the application icon.

message
The text positioned below the `title` in smaller font. If not a string, will use the string representation of the object.

run()
Launch the window. *rumps.Window* instances can be reused to retrieve user input as many times as needed.

Returns a *rumps.rumps.Response* object that contains the text and the button clicked as an integer.

title
The text positioned at the top of the window in larger font. If not a string, will use the string representation of the object.

4.4 Response

class rumps.rumps.**Response**(*clicked*, *text*)
Holds information from user interaction with a *rumps.Window* after it has been closed.

clicked
Return a number representing the button pressed by the user.

The "ok" button will return `1` and the "cancel" button will return `0`. This makes it convenient to write a conditional like,

```
        if response.clicked:
            do_thing_for_ok_pressed()
        else:
            do_thing_for_cancel_pressed()
```

Where *response* is an instance of *rumps.rumps.Response*.

Additional buttons added using methods *rumps.Window.add_button()* and *rumps.Window.add_buttons()* will return 2, 3, ... in the order they were added.

text
Return the text collected from the user.

4.5 Timer

class rumps.**Timer**(*callback*, *interval*)
Python abstraction of an Objective-C event timer in a new thread for application. Controls the callback function, interval, and starting/stopping the run loop.

Changed in version 0.2.0: Method *__call__* removed.

Parameters

- **callback** – Function that should be called every *interval* seconds. It will be passed this *rumps.Timer* object as its only parameter.
- **interval** – The time in seconds to wait before calling the *callback* function.

callback
The current function specified as the callback.

interval
The time in seconds to wait before calling the `callback` function.

is_alive()
Whether the timer thread loop is currently running.

set_callback(*callback*)
Set the function that should be called every `interval` seconds. It will be passed this *rumps.Timer* object as its only parameter.

start()
Start the timer thread loop.

stop()
Stop the timer thread loop.

CHAPTER 5

rumps Functions

5.1 notifications

rumps.**notifications**(*f*)

Decorator for registering a function to serve as a "notification center" for the application. This function will receive the data associated with an incoming OS X notification sent using *rumps.notification()*. This occurs whenever the user clicks on a notification for this application in the OS X Notification Center.

```
@rumps.notifications
def notification_center(info):
    if 'unix' in info:
        print 'i know this'
```

5.2 clicked

rumps.**clicked**(**args*, ***options*)

Decorator for registering a function as a callback for a click action on a *rumps.MenuItem* within the application. The passed *args* must specify an existing path in the main menu. The *rumps.MenuItem* instance at the end of that path will have its *rumps.MenuItem.set_callback()* method called, passing in the decorated function.

Changed in version 0.2.1: Accepts *key* keyword argument.

```
@rumps.clicked('Animal', 'Dog', 'Corgi')
def corgi_button(sender):
    import subprocess
    subprocess.call(['say', '"corgis are the cutest"'])
```

Parameters

- **args** – a series of strings representing the path to a *rumps.MenuItem* in the main menu of the application.
- **key** – a string representing the key shortcut as an alternative means of clicking the menu item.

5.3 timer

rumps.**timer**(*interval*)

Decorator for registering a function as a callback in a new thread. The function will be repeatedly called every *interval* seconds. This decorator accomplishes the same thing as creating a *rumps.Timer* object by using the decorated function and *interval* as parameters and starting it on application launch.

```
@rumps.timer(2)
def repeating_function(sender):
    print 'hi'
```

Parameters **interval** – a number representing the time in seconds before the decorated function should be called.

5.4 timers

rumps.**timers**()

Return a list of all *rumps.Timer* objects. These can be active or inactive.

5.5 application_support

rumps.**application_support**(*name*)

Return the application support folder path for the given *name*, creating it if it doesn't exist.

5.6 notification

rumps.**notification**(*title*, *subtitle*, *message*, *data=None*, *sound=True*)

Send a notification to Notification Center (Mac OS X 10.8+). If running on a version of Mac OS X that does not support notifications, a `RuntimeError` will be raised. Apple says,

> "The userInfo content must be of reasonable serialized size (less than 1k) or an exception will be thrown."

So don't do that!

Parameters

- **title** – text in a larger font.
- **subtitle** – text in a smaller font below the *title*.
- **message** – text representing the body of the notification below the *subtitle*.
- **data** – will be passed to the application's "notification center" (see *rumps.notifications()*) when this notification is clicked.
- **sound** – whether the notification should make a noise when it arrives.

5.7 alert

rumps.**alert**(*title=None*, *message=''*, *ok=None*, *cancel=None*)
Generate a simple alert window.

Changed in version 0.2.0: Providing a *cancel* string will set the button text rather than only using text "Cancel". *title* is no longer a required parameter.

Parameters

- **title** – the text positioned at the top of the window in larger font. If `None`, a default localized title is used. If not `None` or a string, will use the string representation of the object.
- **message** – the text positioned below the *title* in smaller font. If not a string, will use the string representation of the object.
- **ok** – the text for the "ok" button. Must be either a string or `None`. If `None`, a default localized button title will be used.
- **cancel** – the text for the "cancel" button. If a string, the button will have that text. If *cancel* evaluates to `True`, will create a button with text "Cancel". Otherwise, this button will not be created.

Returns a number representing the button pressed. The "ok" button is `1` and "cancel" is `0`.

5.8 debug_mode

rumps.**debug_mode**(*choice*)
Enable/disable printing helpful information for debugging the program. Default is off.

5.9 quit_application

rumps.**quit_application**(*sender=None*)
Quit the application. Some menu item should call this function so that the application can exit gracefully.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

U

V

W