OS X, Python, and the readline module

Intro

One of my active open source projects is called “AOSD” (Apple Open Source Downloader). It is a utility that allows for easily downloading packages that Apple has released on their open source site. I wrote this because I am very interested in the code that Apple releases and have worked with Apple to help keep the releases on that site up-to-date. Last month I rewrote AOSD from scratch to be more than just a hacked together Python script. I wanted to be able to have the tool stand alone and be able to get access to new source updates without needing to rebuild it. As part of this I wrote an interactive command console in it to allow for easier access to the source packages and enable some nice features like being able to tab-complete package names and release/build numbers. This all came together very well, but I was left with a remaining issue of the output of suggested completions becoming sorted. This blog post explains the issues and work-around I needed to use for the implementation of the readline module on OS X.

Background

Python has a module called readline which supports completion and read/write of history files for the Python interpreter. The most significant thing about this module is that the implementation of it differs based on the host OS.

Querying `readline.__doc__` on Ubuntu 14.0.4.3:

```
$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import readline
>>> print(readline.__doc__)
Importing this module enables command line editing using GNU readline.
```

Querying `readline.__doc__` on OS X 10.10.5:

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import readline
>>> print(readline.__doc__)
Importing this module enables command line editing using libedit readline.
```

As you can see, on OS X the implementation of readline is handled by libedit instead of the standard GNU implementation. This is important to make note

of and is mentioned specifically in the Python documentation that the two implementations differ in how they are used.

The following is an example of how to create a custom console application in python:

```
from cmd import Cmd
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print("bleep!")
a = my_cmd()
a.cmdloop()
```

If you were to save this to a file and run it via `python mycmd.py` you would get a prompt that looks like this:

```
$ python blogcmd.py
(Cmd)
```

From here you can enter the command `bleep` and it will print out a new line:

```
(Cmd) bleep
bleep!
(Cmd)
```

Something that you will notice is that if you hit the tab key, it will insert a tab character. While this may be desired behavior when working with text, a command console should map the tab key to assist the user with completing whatever command they are currently typing. This is where the readline module comes in. Modifying the code to add the module and add support for tab completion:

```
from cmd import Cmd
import readline
readline.parse_and_bind("tab: complete")
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print("bleep!")
a = my_cmd()
a.cmdloop()
```

From here running the code again, you will notice one of two things happen when you hit the “tab” key twice:

1. the console suggested two options, “bleep” and “help”
2. the console inserted the tab character

If you experienced the first option, then Python has imported the GNU readline module. However if you experienced the latter of the two, then you are most likely running OS X or have a broken installation of Python.

To get tab-completion to work using readline on OS X you need to issue a different command to the module:

```
from cmd import Cmd
import readline
readline.parse_and_bind("bind ^I rl_complete")
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print("bleep!")
a = my_cmd()
a.cmdloop()
```

Now when running the code again you should be able to hit the tab key twice on the command console and see output like this:

```
$ python blogcmd.py
(Cmd) <tab><tab>
bleep help
(Cmd)
```

This is due to the differing implementation of readline on OS X. If you want to make your Python code take advantage of tab-completion regardless of platform it is being run on you are going to have to implement the following after importing the module:

```
if 'libedit' in readline.__doc__:
    readline.parse_and_bind("bind ^I rl_complete")
else:
    readline.parse_and_bind("tab: complete")
```

This is, unfortunately, the official recommendation of how to check for and handle the different implementations of the readline module in Python.

Armed with this knowledge, you can easily implement your own auto-completion into the command console:

```
from cmd import Cmd
import readline
readline.parse_and_bind("bind ^I rl_complete")
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print("bleep!")
    def complete_bleep(self, text, line, begidx, endidx):
        return ['z', 'a']
a = my_cmd()
a.cmdloop()
```

Running this will result in the following output:

```
$ python blogcmd.py
(Cmd) <tab><tab>
bleep help
(Cmd) ble<tab>
(Cmd) bleep<space><tab><tab>
a z
```

Now you can see here a new method was added to the `my_cmd` class, part of the functionality provided by the `cmd.Cmd` class object is that it allows for completion of available commands as well as the completion for arguments for those commands. You will notice that in the code this was implemented to return the array `['z', 'a']` and in the output on the command line we see it in reverse order `a z`. This is because of the default implementation of readline will sort the output for better readability by us humans.

For most cases this is desired behavior, however when trying to supply completion options for build numbers this can become a problem. Very few build versioning numbers can happily be sorted and preserve the intended ordering of the items.

For example, starting with an array of numbers:

```
['10.0', '10.1', '10.2', '10.3', '10.4', '10.5', '10.6', '10.7', '10.8', '
```

The desired output for this should preserve this exact order (or be inverted, depending on how you want to show chronological releases). However running this through a sorting algorithm will return the ordering as:

```
['10.0', '10.1', '10.10', '10.2', '10.3', '10.4', '10.5', '10.6', '10.7',
```

This was the problem I was running into, where I was returning an array of elements in the desired order but they were getting printed to screen in a different order.

Modifying readline's completion display

The readline module has an API that was introduced in version 2.6 that allows you to assign your own function to perform the action of displaying the completion matches from your command console.

readline.**set_completion_display_matches_hook([*function*])**

> Set or remove the completion display function. If function is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

By using this API, you are able to modify how the readline module will display the output of the completion. So we can modify the code to take advantage of this:

```
from cmd import Cmd
import readline
def match_display_hook(substitution, matches, longest_match_length):
    for match in matches:
        print match
    print readline.get_line_buffer(),
```

```
    readline.redisplay()
readline.parse_and_bind("bind ^I rl_complete")
readline.set_completion_display_matches_hook(match_display_hook)
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print "bleep!"
    def complete_bleep(self, text, line, begidx, endidx):
        return ['z', 'a']
a = my_cmd()
a.cmdloop()
```

Before we test this implementation, there are a couple of things that aren't talked about in the documentation.

Implementing a completion handler

The way the `cmd.Cmd` class works is that it will override the default completion handler from readline to use its own completion handler. This is how it dynamically generates the list of available commands based on the names of the method calls that get implemented on the subclass. It will find any methods that start with `do_` and interpret those as actions, any methods starting with `complete_` with be used as argument completion for the respective command. Each command can have its own completion handler for the arguments specific to that command.

This is what the completion handler on `cmd.Cmd` looks like:

```
def complete(self, text, state):
    """Return the next possible completion for 'text'.

    If a command has not been entered, then complete against command list.
    Otherwise try to call complete_<command> to get list of completions.
    """
    if state == 0:
        import readline
        origline = readline.get_line_buffer()
        line = origline.lstrip()
        stripped = len(origline) - len(line)
        begidx = readline.get_begidx() - stripped
        endidx = readline.get_endidx() - stripped
        if begidx>0:
            cmd, args, foo = self.parseline(line)
            if cmd == '':
                compfunc = self.completedefault                          #
            else:
                try:
                    compfunc = getattr(self, 'complete_' + cmd)          #
                except AttributeError:
                    compfunc = self.completedefault                      #
        else:
            compfunc = self.completenames                                #
        self.completion_matches = compfunc(text, line, begidx, endidx)   #
```

```
    try:
        return self.completion_matches[state]                          #
    except IndexError:
        return None
```

If you look at the comments I've added to this snippet of code, you see that the value that is getting returned from this function is the array of results that get returned from the `complete_` function we have implemented in our `my_cmd` object. This verifies that the output we will see on the display should match the ordering of that of the array we return. However, testing out the implementation yields the same result on OS X:

```
$ python blogcmd.py
(Cmd) <tab><tab>
bleep help
(Cmd) ble<tab>
(Cmd) bleep<space><tab><tab>
a z
```

This is because our custom display hook is not being called at all. You can validate this by adding some `print` debugging statements of your own to the `def match_display_hook(substitution, matches, longest_match_length):` that w

The sorting was coming from within readline

After eliminating our code, and the other Python code we are using, the only thing left is the code of the readline itself. To locate the source of the module, you can open an Python console and query the import:

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import readline
>>> print(readline)
<module 'readline' from '/System/Library/Frameworks/Python.framework/Versi
```

From here we can see that it is being loaded from the system installation of Python 2.7 and running `file` on the path provided shows us that it is a dynamic library:

```
$ file /System/Library/Frameworks/Python.framework/Versions/2.7/lib/python
/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib
/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib
/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib
```

This module is compiled C code, rather than Python, which makes it a little more challenging to debug. Python can load dynamic libraries written in C as modules (you can learn more about it here). Luckily Apple releases the Python implementation they use as part of their open source package release, so we can examine how this particular module is written without having to reverse engineer it from the binary.

```
/* Initialize the module */

PyDoc_STRVAR(doc_module,
"Importing this module enables command line editing using GNU readline.");

#ifdef __APPLE__
PyDoc_STRVAR(doc_module_le,
"Importing this module enables command line editing using libedit readline
#endif /* __APPLE__ */

PyMODINIT_FUNC
initreadline(void)
{
    PyObject *m;

#ifdef __APPLE__
    if (strncmp(rl_library_version, libedit_version_tag, strlen(libedit_ve
        using_libedit_emulation = 1;
    }

    if (using_libedit_emulation)
        m = Py_InitModule4("readline", readline_methods, doc_module_le,
                    (PyObject *)NULL, PYTHON_API_VERSION);
    else

#endif /* __APPLE__ */

    m = Py_InitModule4("readline", readline_methods, doc_module,
                       (PyObject *)NULL, PYTHON_API_VERSION);
    if (m == NULL)
        return;

    ...
    setup_readline();
    ...
}
```

The source shows us that if Apple is compiling this module it is supposed to be initialized taking a slightly different path than if it was compiled otherwise. The doc string for the module is changed to mention the usage of libedit instead of GNU. Following the code, it calls into `setup_readline()`:

```
/* Helper to initialize GNU readline properly. */

static void
setup_readline(void)
{
    ...
    /* Set our completion function */
    rl_attempted_completion_function = flex_complete;
```

```
    ...
}
```

As part of the initial setup, there is a function set to be called when readline attempts to complete any text:

```
/* A more flexible constructor that saves the "begidx" and "endidx"
 * before calling the normal completer */

static char **
flex_complete(const char *text, int start, int end)
{
#ifdef HAVE_RL_COMPLETION_APPEND_CHARACTER
    rl_completion_append_character ='\0';
#endif
#ifdef HAVE_RL_COMPLETION_SUPPRESS_APPEND
    rl_completion_suppress_append = 0;
#endif
    Py_XDECREF(begidx);
    Py_XDECREF(endidx);
    begidx = PyInt_FromLong((long) start);
    endidx = PyInt_FromLong((long) end);
    return completion_matches(text, *on_completion);
}
```

The final line of this function, `return completion_matches(text, *on_completion);` contains a macro when compiled for Apple systems.

```
#ifdef HAVE_RL_COMPLETION_MATCHES
    #define completion_matches(x, y) rl_completion_matches((x), ((rl_compe
#else
    #if defined(_RL_FUNCTION_TYPEDEF)
        extern char **completion_matches(char *, rl_compentry_func_t *);
    #else
        #if !defined(__APPLE__)
            extern char **completion_matches(char *, CPFunction *);
        #endif
    #endif
#endif
```

The macro switches the call of `completion_matches` over to `rl_completion_matches`. When this `readline.c` file is compiled into the readline module it is linked against libedit, where it pulls much of the functionality it uses. By checking the source code for libedit (also released as part of Apple's open source packages), we can see the function that gets called is the following:

```
char **
rl_completion_matches(const char *str, rl_compentry_func_t *fun)
{
    size_t len, max, i, j, min;
    char **list, *match, *a, *b;
```

```
    len = 1;
    max = 10;
    if ((list = el_malloc(max * sizeof(*list))) == NULL)
        return NULL;

    while ((match = (*fun)(str, (int)(len - 1))) != NULL) {
        list[len++] = match;
        if (len == max) {
            char **nl;
            max += 10;
            if ((nl = el_realloc(list, max * sizeof(*nl))) == NULL)
                goto out;
            list = nl;
        }
    }
    if (len == 1)
        goto out;
    list[len] = NULL;
    if (len == 2) {
        if ((list[0] = strdup(list[1])) == NULL)
            goto out;
        return list;
    }
    qsort(&list[1], len - 1, sizeof(*list), _completion_cmp); // we found
    min = SIZE_T_MAX;
    for (i = 1, a = list[i]; i < len - 1; i++, a = b) {
        b = list[i + 1];
        for (j = 0; a[j] && a[j] == b[j]; j++)
            continue;
        if (min > j)
            min = j;
    }
    if (min == 0 && *str) {
        if ((list[0] = strdup(str)) == NULL)
            goto out;
    } else {
        if ((list[0] = el_malloc((min + 1) * sizeof(*list[0]))) == NULL)
            goto out;
        (void)memcpy(list[0], list[1], min);
        list[0][min] = '\0';
    }
    return list;

out:
    el_free(list);
    return NULL;
}
```

This function is quite dense and the terse variable names make it difficult to see what is going on. The function takes the text we have entered on the command line so far as the first argument, then takes the completion

handler we have assigned in python as the second argument. The function will continue to call the completion handler until it returns `NULL`, which signifies there are no more completions available for the given string. It will then sort the completions via a call to `qsort` with a callback that uses `strcmp` and then return the list.

But what about set_completion_display_matches_hook ?

So why is there an API that supposedly allows for modification of the display of matched items?

```
static PyObject *completion_display_matches_hook = NULL;

static PyObject *
set_completion_display_matches_hook(PyObject *self, PyObject *args)
{
    PyObject *result = set_hook("completion_display_matches_hook", &comple
    #ifdef HAVE_RL_COMPLETION_DISPLAY_MATCHES_HOOK
        /* We cannot set this hook globally, since it replaces the default
        rl_completion_display_matches_hook = completion_display_matches_ho
        #if defined(_RL_FUNCTION_TYPEDEF)
            (rl_compdisp_func_t *)on_completion_display_matches_hook : 0;
        #else
            (VFunction *)on_completion_display_matches_hook : 0;
        #endif
    #endif
    return result;
}
```

This is the call that gets made when you assign a function to be the display hook. This will successfully assign the new function to be the hook for displaying the output, however the implementation of libedit never checks for this hook. The GNU implementation of readline will perform a check to see if this hook is assigned and then call the hook and return immediate afterwards. Since this check is skipped in libedit, the call for displaying the matched output is sent directly to `fn_display_match_list`:

```
fn_display_match_list (EditLine *el, char **matches, size_t num, size_t wi
{
    size_t line, lines, col, cols, thisguy;
    int screenwidth = el->el_terminal.t_size.h;

    /* Ignore matches[0]. Avoid 1-based array logic below. */
    matches++;
    num--;

    /*
     * Find out how many entries can be put on one line; count
     * with one space between strings the same way it's printed.
     */
    cols = (size_t)screenwidth / (width + 1);
    if (cols == 0)
```

```
        cols = 1;

    /* how many lines of output, rounded up */
    lines = (num + cols - 1) / cols;

    /* Sort the items. */
    qsort(matches, num, sizeof(char *), _fn_qsort_string_compare); // anot

    /*
     * On the ith line print elements i, i+lines, i+lines*2, etc.
     */
    for (line = 0; line < lines; line++) {
        for (col = 0; col < cols; col++) {
            thisguy = line + col * lines;
            if (thisguy >= num)
                break;
            (void)fprintf(el->el_outfile, "%s%-*s", col == 0 ? "" : " ", (
        }
        (void)fprintf(el->el_outfile, "\n");
    }
}
```

This function parses through the list of matched items and will then format them in columns to display on the terminal. Before displaying it will sort them using `qsort`, but this time with a callback using `strcasecmp` instead of `strcmp`.

The Workaround

Due to how readline is implemented via libedit on OS X, you have only two options:

1. install the GNU readline and use that over the system version
2. make the readline module use a different version of libedit

I would strongly recommend everyone follow the first option.

If that isn't a viable solution, you *can* create a work-around by shipping your own implementation of the readline module that links to your own copy of libedit.

How to override the system readline module

First off you need to grab the source of libedit and make the modifications you want. You can grab the source from Apple here or you can grab it already modified to remove sorting here. As of this post the latest release of libedit is 40, and builds successfully from source. If you are building from Apple's source and not the modified version you may also need to change the install name of the library so dyld will not ignore loading it if libedit is already loaded in-process.

Once you have built libedit you will need to grab a copy of the existing `readline.so` file and copy it to the same directory as your Python script file. From here you will need to change the link path on `readline.so` from from `/usr/lib/libedit.3.dylib` to instead point to `@loader_path/libedit-unsorted.3.dylib`. After doing this you can put the `libedit-unsorted.3.dylib` (or whatever you named it, the link path and name must match) next to the `readline.so` file.

Now your directory should look like this:

```
$ ls -1
blogcmd.py
libedit-unsorted.3.dylib
readline.so
```

If you go back to the source code of your Python script, it should look like this:

```
from cmd import Cmd
import readline
readline.parse_and_bind("bind ^I rl_complete")
class my_cmd(Cmd,object):
    def do_bleep(self, s):
        print "bleep!"
    def complete_bleep(self, text, line, begidx, endidx):
        return ['z', 'a']
a = my_cmd()
a.cmdloop()
```

When executed, you should see the following results:

```
$ python blogcmd.py
(Cmd) <tab><tab>
bleep help
(Cmd) bleep<space><tab><tab>
z a
(Cmd) bleep
```

The completion display output now matches the order of the items returned from the `complete_bleep` method call!

Integrating this into a module

While this works, you may have some trouble trying to integrate this into a module that can be installed and used. The way I have implemented a work-around for this is to also ship your own implementation of the `cmd.Cmd` class. I will then use the following check to selectively import the system `cmd` module or the custom one.

```
import sys
import platform
if sys.platform == 'darwin' and not 'ppc' in platform.machine(): # because
```

```
     from .readline_unsorted.cmd import *
else:
    from cmd import Cmd
```

I have a custom module that gets imported that only contains a slightly modified copy of the system `cmd.py` file, `readline.so`, and `libedit.dylib` (as modified above). The modifications I have applied to the `cmd.py` file are as follows:

1. remove all instances of the line `import readline` that are spaced throughout the file (there should be 3)
2. adding the following above the `import string` line

```
import readline
readline.parse_and_bind("bind ^I rl_complete")
```

Now your completion display output should match the ordering that was returned from your `complete_` handler.

Final Thoughts

This post was a product of spending a the entire day trying to divine the cause of the sorted output and why the defined APIs were not being called. I find it hard to believe I am the first person that as come across the `set_completion_display_matches_hook` API not working as intended (maybe it is a regression), but it does seem that I am the first person to document anything about it in relation to the libedit implementation of readline. Hopefully this saves someone else the headache of trying to work this out for themselves.

Thanks to Michael Lynn for helping me with this.
