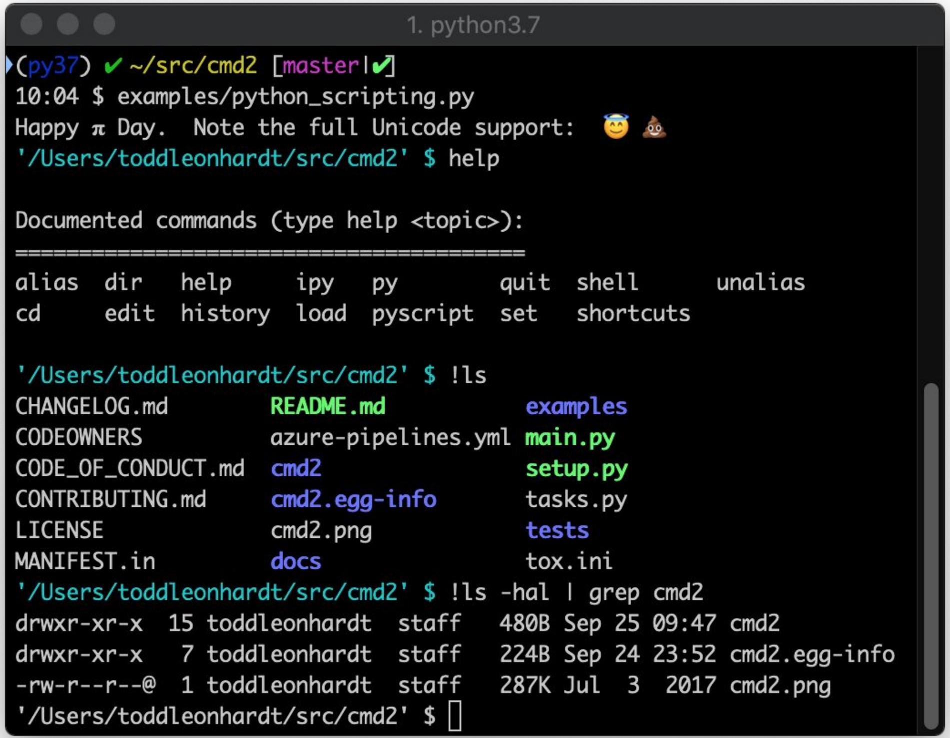
A Hands-on Guide to Building Interactive Command-Line Apps with cmd2

Todd Leonhardt - todd.leonhardt@gmail.com
Kevin Van Brunt - kmvanbrunt@gmail.com

Overview

- What is cmd2?
- Why would you want to use it?
- What can it do for you?
- How do you use it?
- Where can you learn more?



```
(py37) ✔ ~/src/cmd2 [master|✔]
10:04 $ examples/python_scripting.py
Happy π Day.  Note the full Unicode support:  😇 💩
'/Users/toddleonhardt/src/cmd2' $ help

Documented commands (type help <topic>):
========================================
alias  dir   help     ipy   py        quit  shell      unalias
cd     edit  history  load  pyscript  set   shortcuts

'/Users/toddleonhardt/src/cmd2' $ !ls
CHANGELOG.md        README.md           examples
CODEOWNERS          azure-pipelines.yml main.py
CODE_OF_CONDUCT.md  cmd2                setup.py
CONTRIBUTING.md     cmd2.egg-info       tasks.py
LICENSE             cmd2.png            tests
MANIFEST.in         docs                tox.ini
'/Users/toddleonhardt/src/cmd2' $ !ls -hal | grep cmd2
drwxr-xr-x  15 toddleonhardt  staff   480B Sep 25 09:47 cmd2
drwxr-xr-x   7 toddleonhardt  staff   224B Sep 24 23:52 cmd2.egg-info
-rw-r--r--@  1 toddleonhardt  staff   287K Jul  3  2017 cmd2.png
'/Users/toddleonhardt/src/cmd2' $ 
```

Example code for this presentation is available on GitHub:
https://github.com/python-cmd2/talks/tree/master/PyOhio_2019

What is cmd2?

- Python module for building interactive command-line applications
  - Command line **interpreters** (CLI)
    - Other names **shell**, **console**, **REPL**
    - Examples: bash, ipython, sqlite3
- Extends Python’s built-in cmd module
  - cmd2 adds a ton of useful features on top of cmd
    - To save you schedule and budget
- Open-source available on GitHub
  - You can help make it even better
    - Welcoming to new contributors - good Contributor’s Guide
  - Permissive MIT license
    - Suitable for proprietary commercial applications as well as open-source

What cmd2 is NOT used for

- Building command-line utilities (CLU)
  - Accept arguments at invocation, execute, and terminate
  - Examples: ls, grep, git
  - Python's built-in argparse module or Click have this well covered
  - Actually … cmd2 can be used for this use case via arguments at invocation

- Building text user interfaces (TUI)
  - Full-screen applications more like a GUI but in a terminal
  - Examples: vim, emacs, top
  - Python's built-in curses module can be used for this

Why build a CLI?

- Advantages of a CLI over a GUI
  - Easier and faster to write
  - Scriptable
  - Easily interact with standard CLU tools
  - Can be executed over a remote non-graphical connection
    - SSH
- Advantages of a CLI over a CLU
  - More user friendly
  - Ability to be interactive
  - Can maintain a persistent connection to stuff
    - And keep state between commands
- CLI apps are very common in certain domains
  - DevOps / Automation
  - Security

Why build a CLI in Python? 🐍

- Developer productivity
  - Solve same problem in ⅓ to ¼ time it takes in Java, C/C++, or C#
- Cross-platform
  - Your app will run on Windows, Mac, and Linux
  - With a little care your app can be truly cross-platform
- Easily leverage existing C/C++ in Python
  - Cython
  - CFFI
  - SWIG
- Amazing libraries
  - boto3, requests, simplejson, jsonschema, SQLAlchemy, cryptography, lxml, numpy, pandas

Why use cmd2 to build your CLI?

- Customers often ask for CLI tools with features such as:
  - Scriptable with Python
  - Good tab completion
  - Built-in help for all commands
  - Full unicode support for command arguments and output
  - History of commands executed
  - Command output redirectable to file
  - Command output pipeable to shell command
- cmd2 meets all of these needs
  - And a whole lot more
- Alternatives to cmd2
  - Python Prompt Toolkit

cmd2 Main Features

Features inherited from cmd:

- Easy application creation
  - Inherit from **Cmd** class
  - **cmdloop** method handles REPL
- Easy to add commands
  - **do_foo** method creates **foo** command
- Help for **foo** command is provided by
  - **help_foo** method if it exists
  - **do_foo** docstring if it doesn't
- Tab completion of command names
  - Automatic
- Tab completion of command arguments
  - **complete_foo** method
- Application lifecycle hooks
  - **precmd**, **postcmd**
  - **preloop**, **postloop**

Features added by cmd2:

- Text file scripting of your application
  - **run_script** command and **@** shortcut
- Python scripting of your application
  - **run_pyscript** command
- Run shell commands
  - **shell** command and **!** shortcut
- **|** pipes command output to shell command
- Redirect command output to file
  - **>** overwrites, **>>** appends
  - Bare **>** or **>>** redirect to clipboard
- Searchable command history
  - Readline up/down arrows and <Ctrl>+r
  - **history** command
- Argparse decorators for argument parsing
- Amazing tab completion
- Command aliases and macros
- Startup script
- Embedded Python shell, (optional) IPython
- Regression testing framework

Create a cmd2 application

- Building a cmd2 application is simple
    - Import **cmd2**
    - Create a class which inherits from **cmd2.Cmd**
        - Make additions, changes, and deletions as desired
    - Instantiate an object of the class you created
    - Call the **cmdloop()** method on this object

Create a cmd2 application (code)

```
#!/usr/bin/env python3
import cmd2
from cmd2 import style

class BasicApp(cmd2.Cmd):
    CUSTOM_CATEGORY = 'My Custom Commands'
    def __init__(self):
        super().__init__(persistent_history_file='cmd2_history.dat',
                         startup_script='scripts/startup.txt', use_ipython=True)
        self.intro = style('Welcome to PyOhio 2019 and cmd2!', fg='red') + ' 😀'
        # Allow access to your application in py and ipy via self
        self.locals_in_py = True

        # Set the default category name
        self.default_category = 'cmd2 Built-in Commands'


if __name__ == '__main__':
    app = BasicApp()
    app.cmdloop()
```

Add a cmd2 command

- Adding a cmd2 command is easy
  - Create a method within your class which inherits from **cmd2.Cmd**
  - Give this method a name which begins with **do_**
    - A method **do_foo** results in a command **foo**
  - Help for this command will be provided by the method's docstring

Add a cmd2 command (code)

```
#!/usr/bin/env python3
import cmd2
from cmd2 import style

class BasicApp(cmd2.Cmd):
    CUSTOM_CATEGORY = 'My Custom Commands'
    def __init__(self):
        super().__init__(persistent_history_file='cmd2_history.dat',
                         startup_script='scripts/startup.txt', use_ipython=True)
        self.intro = style('Welcome to PyOhio 2019 and cmd2!', fg='red') + ' 😀'
        # Allow access to your application in py and ipy via self
        self.locals_in_py = True

        # Set the default category name
        self.default_category = 'cmd2 Built-in Commands'

    @cmd2.with_category(CUSTOM_CATEGORY)
    def do_intro(self, _):
        """Display the intro banner"""
        self.poutput(self.intro)

if __name__ == '__main__':
    app = BasicApp()
    app.cmdloop()
```

Add help for a cmd2 command

- Adding help for a cmd2 command is trivial
  - By default, the method docstring **is** the help
- More complicated and/or dynamic help is possible
  - Add a **help_foo** method corresponding to the do_foo method
    - If present, the **help_foo** method provides help instead of the docstring
- Argparse-based commands use the parser’s help output

Tab completion for a cmd2 command

- cmd2 automatically supports tab completion of command names
- It is easy to add support for custom tab-completion of command arguments
  - Add a **complete_foo** method corresponding to the do_foo method
    - If present, the complete_foo method provides readline-style tab completion for command arguments to this command
- cmd2 has built-in methods which provide tab completion of file system paths, directories, and executables
  - These are used by built-in commands when it makes sense
  - You can use these completion methods with your custom commands
  - Whenever paths are supported, both relative and absolute paths are OK
    - User path expansion is automatic (so using ~/ is fine)
- cmd2 has other/better options for tab completion when using one of the argparse decorators

Searchable command history

- **<Ctrl>+r** works just like it does for a reverse-search in Bash
- Arrow keys also work the same way as in Bash
- **history** command supports display and searching history in a flexible way
    - no argument - displays entire command history
    - integer argument - displays single history item, by index (1-based)
    - a..b, a:b, a:, ..b - list history items by a span of indices (inclusive)
    - string argument - display all commands matching string search
    - arg is /enclosed in forward-slashes/ - regular expression search
- **history -r** option re-runs previous command(s)
- **history -o** option saves command(s) from history to a file
- **history -e** option allows you to edit and then execute previous command(s)
- **history -t** option generates a transcript file for regression testing
- **history -c** clears all history

Built-in editor

- **edit** command edits a file
    - string argument - edits a file at this path
- The editor used is determined by the *editor* settable parameter
    - Can be changed using the **set** command
        - set editor emacs

Settable environment parameters

- **set** command used to change settings values as well as display them
  - set -l
    - Shows settable parameters in long form (with descriptions)
  - set param value
    - Sets parameter “param” to have value “value”
    - e.g.: set editor emacs
  - Good tab completion will show you options and tab complete param names
- Easy to add or remove settable parameters in your app
  - Just edit the **settable** dictionary attribute in your cmd2.Cmd class

Shell commands

- Run any OS shell command on your path within your app
  - with **shell** command or the **!** shortcut
  - e.g. !ls
- shell command supports good tab-completion
  - Initially of shell command names
  - Subsequently of file system paths

Output redirection or pipe

- **Output redirection** just like in POSIX shells
  - Use **>** to overwrite a file
  - Use **>>** to append to a file
  - If file is left blank, output redirected to clipboard
- Pipe command output to shell command
  - Use **|** to pipe command output to a shell command
- Some limitations apply
  - Can only redirect or pipe to shell commands, not other cmd2 application commands
    - However multiple redirection and/or pipes are supported
  - stdout gets redirected/piped, stderr does not
- Usefulness of these features cannot be overstated
  - Limited only by the creativity and command-line knowledge of end users
  - Instant access to grep, less, more, awk, sed, wc, sort, etc.
- Fully cross-platform (works the same on Windows)

Commands at invocation

- Treat CLI args to your cmd2 app as cmd2 commands
    - e.g. “./basic.py ‘echo foo’”
- If there are multiple commands, then you should enclose each command along with its arguments in quotes
    - ./basic.py “cmd1 arg1” “cmd2 arg2”
    - Can also use single quotes instead of double quotes
- Allows you to pre-execute commands before entering the interactive prompt
    - Startup script runs before any commands at invocation
- Enables you to finish with a **quit** command so your CLI runs like a CLU
    - Which means it can be scripted from an external application

Text File Scripts

- Run scripts stored in ASCII or UTF-8 text files
  - One command per line
    - Exactly as you would type it within the app
- Use the **run_script** command or **@** shortcut
  - e.g. @my_script.txt
- Comments are supported for documenting scripts
  - Python style - lines starting with “#” are comments
- **run_script** command supports tab-completion of file system paths
- **_relative_run_script** command or **@@** shortcut
  - Variant for use within a script which uses paths relative to the first script

Aliases

- A command that enables replacement of a word by another string
    - alias create lh !ls -hal
        - lh /home/user → !ls -hal /home/user
- Redirectors and pipes should be quoted in alias definition to prevent the alias create command from being redirected
    - alias create save_results print_results ">" out.txt
        - save_results → print_results > out.txt
- Tab completion recognizes an alias, and completes as if its actual value was on the command line

Macros

- Similar to an alias, but it can contain argument placeholders
  - macro create backup !cp "{1}" "{1}.ver_{2}"
    - backup file.txt 4 → shell cp "file.txt" "file.txt.ver_4"
- Like aliases, pipes and redirectors need to be quoted in the definition
  - macro create lc !cat "{1}" "|" less
    - lc file.txt → shell cat "file.txt" | less
- To use the literal string {1} in your command, escape it this way: {{1}}
- Because macros do not resolve until after hitting Enter, tab completion will only complete paths while typing a macro.

Argument Processing with argparse-based decorators

- Three decorators to parse arguments and change args passed to command
  - **@with_argparser**
    - Accepts an instance of argparse.ArgumentParser (or class derived from it)
    - Command gets passed parsed arguments as an argparse.Namespace
  - **@with_argparser**_and_unknown_args
    - Accepts an instance of argparse.ArgumentParser (or class derived from it)
    - Command gets passed parsed arguments plus a list of unknown/unparsed arguments
  - **@with_arg**ument_list
    - No arguments required for decorator
    - Arguments get parsed using shlex and command gets passed a list of arguments

Tab completion using argparse decorators

- Automatic tab completion of flag names
- Tab completion of arguments values specified by one of five parameters to parsers.add_argument()
    - choices
    - choices_function / choices_method
    - completer_function / completer_method
- Keeps track of state
- Displays current argument hint when no completion results exist
- CompletionItem class provides context for completion results
- Standard argparse.nargs values augmented with range tuple capability
    - nargs=(5,) - accept 5 or more items
    - nargs=(8, 12) - accept 8 to 12 items

Embedded Python interpreter

- **py** command will run its arguments as a Python command
    - Without arguments, it enters an interactive Python session with tab completion and history
- **py** can call “back” to your application with **app(‘command args’)**
        - Just like Python scripts run via **run_pyscript**
- Through **self** it has full access to your application instance
- Any variables changed or created will persist for the life of the application
    - py x = 5
    - py print(x)
- Can also run Python scripts via **py run(‘myscript.py’)**
    - But can’t pass arguments to these scripts like you can with **run_pyscript**
- What is it good for?
    - Interactively developing scripts to run with **run_pyscript**

Embedded IPython interpreter (opt-in)

- **ipy** command available if cmd2.Cmd class is instantiated with **ipython=True**
  - Only available if IPython is installed on your system
- The **ipy** command enters an embedded interactive IPython session
  - Can access everything within your application from here via **self**
  - Any changes made to your app via **self** will persist
    - However any local or global variable created will not persist
- You cannot call cmd2 commands within this embedded IPython shell
- IPython is much more user friendly than basic Python shell
  - Good tab-completion
  - Get help on objects and functions with **?**
  - Good built-in debugger
- What is it good for?
  - Debugging and development - comprehensive object introspection
  - Data analysis - full power of Python scientific stack to analyze command output/results

Python scripts

- Run python scripts within your app using the **run_pyscript** command
  - e.g.: run_pyscript myscript.py foo bar ‘baz 23’
    - Runs “myscript.py” Python script and passes it the arguments [‘foo’, ‘bar’, ‘baz 23’]
- Python scripts can run cmd2 commands
  - **app(‘command args’)**
- Python scripts have access to full cmd2 application via **self**
  - Allows conditional control flow logic in Python scripts
    - Where behavior depends on result of previous command(s)
- Scripts have full access to all Python modules installed on your system
- Python scripts are MUCH more powerful and flexible than text-file scripts
  - Conditional control flow
  - Access to the entire world of generic Python outside of your cmd2 app
- See pyscript_example.py example and conditional_flow.py script for more info

Asynchronous Alerting

- Asynchronous feedback to the user without interfering with the command line
  - async_alert()
    - Display an important message to the user while they are at the prompt in between commands. To the user it appears as if an alert message is printed above the prompt and their current input text and cursor location is left alone.
  - async_update_prompt()
    - Update the prompt while the user is still typing at it. This is good for alerting the user to system changes dynamically in between commands. For instance you could alter the color of the prompt to indicate a system status or increase a counter to report an event.
  - set_window_title()
    - Set the terminal window title
- See async_printing.py example for more info

Full Unicode support

- Full Unicode support
  - Command names
  - Command arguments
  - Command output
  - Intro banner
  - Prompt
- UTF-8 encoding is used by default
- Some limitations apply
  - Parser treats following symbols special: “ ‘ | > >> # ;
- Why should you care?
  - Support for command arguments and output in any language, not just English
    - Chinese, Spanish, Hindi, Arabic, Portuguese, Bengali, Russian, Emjoi, etc.

Compatibility

- OS cross-platform compatibility
  - Features work the same on Windows, macOS, and Linux
  - An out-of-the-box cmd2 application is a better shell than cmd.exe on Windows!
    - Due to better tab completion and Unix shell features
- Python version compatibility
  - Support Python 3.5+
  - Features work the same on all versions of Python
- Continuous Integration ensures both compatibility and stability
  - Every PR runs all unit tests on multiple versions of Python on Linux, Windows, and Mac
  - Code coverage of unit tests is > 97%, so features are very stable
  - Use flake8 for static analysis
  - Ensure our documentation builds without warnings
  - All of this helps make it easy for new contributors to submit a Pull Request (PR)
    - They don’t have to worry that they might have broken something

Built-in regression testing framework

- Called “transcript testing”
- A **transcript** combines a command and its expected output
- Any regular expression (regex) can be used within the expected output
  - By putting the regex with forward slashes /.../
- Automatic transcript generation
  - **history -t**
  - **run_script -t**
- Run transcript test from command line
  - ./basic.py -t scripts/basic.transcript

Application Lifecycle Hooks

- Functions exist to implement behavior at every step of the application and command lifecycle
- Application
  - **preloop** - executed once when cmdloop is called
  - **postloop** - executed once when cmdloop is about to return
- Command
  - **Postparsing hooks**
    - called after the user input has been parsed but before execution of the command
  - **Precommand hooks**
    - called after postparsing hooks and after any output redirection has begun
  - **Postcommand hooks**
    - called once the command method has returned but while output still redirected
  - **Command finalization hooks**
    - called even if one of the other types of hooks or the command method raise an exception

Where to learn more

- Github
  - https://github.com/python-cmd2/cmd2
- Documentation:
  - http://cmd2.readthedocs.io/en/latest/
- Examples
  - https://github.com/python-cmd2/cmd2/tree/master/examples
- Videos
  - Look for the video from this presentation, others that exist are out of date

Last but not least ...

- Todd works at [Amazon Web Services]() (AWS) on EC2 Core Services team
  - Located in Herndon, VA; Seattle, WA; and Cape Town, South Africa
  - We are always looking for smart, motivated software developers to join our team
  - Programming typically in a mix of Java and Python or Ruby
- Kevin works at [Research Innovations, Inc]() (RII)
  - Located Arlington, VA and Melbourne, FL
  - Need to be a U.S. citizen
  - Looking for software developers and cyber-security professionals
  - On the developer side, strong preference for proficiency in both C/C++ and Python
- Please get in touch with either of us if you would like to know more