Prompt Toolkit Action Completer Documentation

Release 1.1.1

Stephen Bunn <stephen@bunn.io>

Sep 24, 2020

CONTENTS

```
from pathlib import Path

from action_completer import ActionCompleter
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.completion import PathCompleter
from prompt_toolkit.validation import Validator

completer = ActionCompleter()


@completer.action("cat")
@completer.param(
   PathCompleter(),
   cast=Path,
   validators=[
      Validator.from_callable(
         lambda p: Path(p).is_file(),
         error_message="Path is not an existing file"
      )
   ]
)
def _cat_action(filepath: Path):
   with filepath.open("r") as file_handle:
      print(file_handle.read())


prompt_result = prompt(
   ">>> ",
   completer=completer,
   validator=completer.get_validator()
)
completer.run_action(prompt_result)
```

To get started using this package, please see the *Getting Started* page!

CHAPTER

ONE

GETTING STARTED

Welcome to Action Completer!
This page should hopefully provide you with enough information to get you started with defining actions, groups, and parameters for use with prompt-toolkit.

1.1 Installation and Setup

Installing the package should be super duper simple as we utilize Python's setuptools.

```
$ poetry add prompt-toolkit-action-completer
$ # or if you're old school...
$ pip install prompt-toolkit-action-completer
```

Or you can build and install the package from the git repo.

```
$ git clone https://github.com/stephen-bunn/prompt-toolkit-action-completer.git
$ cd ./prompt-toolkit-action-completer
$ poetry build
$ pip install ./dist/*.whl
```

1.2 Usage

This package supplies both a *ActionCompleter* and a *ActionValidator* for direct use with prompt-toolkit. You should be passing instances of the *ActionCompleter* directly to the call to `prompt()` as the `completer` keyword argument.

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

prompt(">>> ", completer=completer)
```

1.2.1 Defining Actions

This of course isn't very useful right now since we haven't registered any actions to the completer. We can easily add a simple `hello` action by decorating a callable **before** we make the call to `prompt()`:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello")
def _hello_action():
    print("Hello, World!")


prompt(">>> ", completer=completer)
```

With this little bit of logic we will automatically get `hello` completions from our prompt call.

Still not very useful though. We really want to be able to determine what action to execute based on the output of our prompt call. Luckily, the completer can take whatever output the prompt has produced and determine what action should be called.

We do this by using a new method on the completer, *`run_action()`*. We can simply give this method the text produced by the prompt and it will do it's best to execute the desired callable and return whatever value the registered action returns:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello")
def _hello_action():
    print("Hello, World!")


prompt_result = prompt(">>> ", completer=completer)
completer.run_action(prompt_result)
```

Now we have something that is fairly useful. Automatic completion and execution of some registered callable. However, we will start to run into issues with the execution of the action callable when the user starts providing inputs that the callable either isn't expecting or can't handle. To protect against this, we can use the *`ActionValidator`* to validate the prompt buffer state before we attempt to execute the action.

Because the validator is a custom validator that depends on the state of the completer, it's recommended that you use another little helper method accessible right off of the completer instance, *`get_validator()`*. This helper method will give you a new instance of *`ActionValidator`* that will be able to check that the current prompt can be adequately handled by the registered action.

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello")
def _hello_action():
    print("Hello, World!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

That is the very basics of using the *ActionCompleter*, with this you can easily get started creating some basic tools. But we can get a lot more detailed and provide even more useful features by also providing completion for action parameters!

1.2.2 Parameter Completion

These actions won't be terribly useful unless we can supply some user specific inputs. In order to help complete these parameters we need to also register the completable parameters with the action we decorated. We can easily do this using the *param()* decorator:

```
@completer.action("hello")
@completer.param(None)
def _hello_action(name: str):
   print(f"Hello, {name!s}!")

prompt(">>> ", completer=completer)
```

You can now see that we are allowing for a parameter that is automatically provided to the executed callable as the first parameter! We are giving the *param()* decorator a `source` of `None` to indicate that we don't necessarily want completion enabled for this parameter.

Completion Sources

There are many different available sources (some more useful than others). These completion sources are the powerhouse of how parameters are selected by the user.

None (`None`)

None completions indicate that a parameter is required and gives you all the nifty features of an action parameter, but doesn't attempt to do any real completion.

Warning: A fairly big caveat of parameter completion for `None` source inputs, is that we don't support values containing spaces as properly handled inputs by the user. Values containing spaces are very difficult to distinguish from upcoming parameters. We do support the ability for users to escape spaces to include them in their parameter input.

```
>>> hello Stephen\ Bunn
Hello, Stephen Bunn!
```

However, this is fairly tedious and error-prone for the customer, so we recommend that you use other action parameter completion sources if you intend for the completable values to contain spaces.

Basic (`str`)

A basic string can be provided as a completion source that will force the value of the parameter to always be whatever the value of this string is. Might not seem very useful, but I've run into several situations when building dynamic actions that this can help out with.

```
@completer.action("hello")
@completer.param("world")
def _hello_world_action(value: str):
    print(f"Hello, {value.title()!s}")
```

Iterable (`List[str]`)

A list of strings can be provided to allow for multiple selections the user can make for the parameter. As long as the value they give is in this list of strings, the input will be considered valid and will be passed through to the action.

This source is a good substitute if you want to be able to complete `enum.Enum` value and cast the value back to the enum instance in the action. Since prompt deals only with strings (not enum values), it is easier for you to handle that decomposition and casting from an enum yourself.

```
@completer.action("hello")
@completer.param(["Mark", "John", "William"])
def _hello_person_action(name: str):
    print(f"Hello, {name!s}")
```

Completer (`Completer`)

One of the more useful parameter sources is another `Completer` instance. We will give the appropriate parameter value to the nested completer and any completion that the completer determines should be yielded will be yielded.

```
from prompt_toolkit.completion import PathCompleter

@completer.action("cat")
@completer.param(PathCompleter())
def _cat_action(filepath: str):
   with open(filepath, "r") as file_handle:
      print(file_handle.read())
```

Callable (`Callable[[Action, ActionParam, str], Iterable[str]]`)

Another useful parameter source is just a custom callable. This callable should return some kind of iterable of strings that should be considered completion results. As inputs, this callable will take the following positional arguments:

- `Action` (*Action*) - The action that is being triggered
- `ActionParam` (*ActionParam*) - The associated action parameter that is requesting completions
- `str` (`str`) - The current value of the action parameter

Whatever list, tuple, or generator of strings is returned will be used as the completion results.

```
def _get_completions(action: Action, param: ActionParam, value: sr) -> Iterable[str]:
   return [str(value) for value in range(12)]

@completer.action("hello")
@completer.param(_get_completions)
def _hello_dynamic_action(dynamic_value: str):
   print(f"Hello, {dynamic_value!s}")
```

1.2.3 Parameter Casting

It gets pretty tedious to have to manually cast our parameter results from strings into custom data types in the action itself. So give the `cast` keyword argument to the *param()* decorator and we will do it for you automatically.

```
@completer.action("x2")
@completer.param(["1", "2", "3"], cast=int)
def _x2_action(num: int):
   print(f"Number {num!s} times 2 is {num * 2!s}")
```

Note, that we don't do anything clever when casting this value. If you request us to cast the parameter to an `int` and the string contains alpha characters, it will fail with the traditional `ValueError`. To avoid this situation, continue reading on through to handling parameter validation.

1.2.4 Parameter Validation

Going back to our `cat` action example in *Completer (Completer)*, we can greatly improve the usability of this by giving the action parameter some validation! Simply pass a `Validator` instance as a value in the `validators` keyword argument, and you can ensure that the completed path is an existing file.

```
from pathlib import Path

from prompt_toolkit.completion import PathCompleter
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.validation import Validator

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("cat")
@completer.param(
    PathCompleter(),
    cast=Path,
    validators=[
        Validator.from_callable(
            lambda p: Path(p).is_file(), error_message="Path is not an existing file"
        )
    ],
)
def _cat_action(filepath: Path):
    with filepath.open("r") as file_handle:
        print(file_handle.read())


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Because your parameter validation *may* require that you also take into consideration the context of the parameter, you can also pass a custom validator callable. Similar to the custom callable parameter completion sources available in *Callable (Callable[[Action, ActionParam, str], Iterable[str]])*, you can also pass a callable with a specific signature to the `validators` list. As inputs, this callable will take the following positional arguments:

- `ActionParam` (*ActionParam*) - The associated action parameter that needs to be validated
- `str` (`str`) - The current value of the action parameter
- `List[str]` (List[`str`]) - The previously extracted prompt buffer fragments

This callable should raise a `ValidationError` when validation fails. For example, let's create an action that creates a new `.txt` file in a specified directory and verifies that the file can safely be created:

```
from pathlib import Path
from typing import List

from prompt_toolkit.completion import PathCompleter
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.validation import ValidationError, Validator

```

(continues on next page)

(continued from previous page)

```
from action_completer import ActionCompleter, ActionParam

completer = ActionCompleter()


def _validate_touch(action_param: ActionParam, param_value: str, fragments:␣
↪List[str]):
    dirpath = Path(fragments[0])
    filepath = dirpath.joinpath(f"{param_value!s}.txt")
    if filepath.is_file():
        raise ValidationError(message=f"File at {filepath!s} already exists")


@completer.action("touch")
@completer.param(
    PathCompleter(only_directories=True),
    cast=Path,
    validators=[
        Validator.from_callable(
            lambda p: Path(p).is_dir(), error_message="Not an existing directory"
        )
    ],
)
@completer.param(None, validators=[_validate_touch])
def _touch_txt_action(dirpath: Path, name: str):
    filepath = dirpath.joinpath(f"{name!s}.txt")
    with filepath.open("r") as file_handle:
        print(file_handle.read())


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

By default, the *ActionValidator* will validate that you are providing the **exact** amount of parameters for an action. If you need to allow for the user to enter additional text into the prompt past the defined parameters, you can set the `capture_all` flag on the action to `True`. This will disable the check for an exact number of parameters and will instead ensure that the user gives at least the number of defined parameters.

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello", capture_all=True)
@completer.param(["Mark", "John", "William"])
def _hello_name(name: str, *args):
    print(f"Hello, {name!s}!")
    print(f"Additional: {args!s}")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
```

(continues on next page)

(continued from previous page)

```
completer.run_action(prompt_result)
```

1.2.5 Nested Groups

Likely you will run into the situation where you want to create a subgroup of actions under a specific completable name. You can do this fairly easily by making use of the *group()* method which will allow you to define a subgroup on an existing group.

Any related actions (or even additional nested subgroups) should be created by using the *action()* or *group()* methods from the newly created *ActionGroup* instance.

```
from pathlib import Path

from prompt_toolkit.completion import PathCompleter
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.validation import Validator

from action_completer import ActionCompleter

completer = ActionCompleter()

hello_group = completer.group("hello")


@hello_group.action("world")
def _hello_world():
    print("Hello, World!")


@hello_group.action("custom")
@completer.param(None)
def _hello_custom(name: str):
    print(f"Hello, {name!s}!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Important: Note that the *param()* decorator is only available from the *ActionCompleter* instance. You should always be using the completers instance that is sent to the `prompt()` call to register any parameters for **any** actions (no matter how nested they are).

```
nested_group = completer.group("nested-group")

# Invalid, @param is not available on the group
@nested_group.action("invalid-action")
@nested_group.param(None)
def _invalid_action(invalid_param):
    ...
```

(continues on next page)

(continued from previous page)

```
# Valid, always use @param from the completer
@nested_group.action("valid-action")
@completer.param(None)
def _valid_action(valid_param):
    ...
```

1.2.6 Styling Completions

Now that we have some context into how we create groups, actions, and parameters, we can talk about customizing the style of the completions. Each *ActionGroup*, *Action*, and *ActionParam* has the `style`, `selected_style`, `display`, and `display_meta` properties.

Color

If you want to change the coloring of the completion results, you can use the `style` and `selected_style` properties to do so. These styles are given directly to the `Completion` instance and will result in the completion results being styled all the same way:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello")
@completer.param(
    ["Mark", "John", "William"],
    style="fg:white bg:red",
    selected_style="fg:red bg:white bold",
)
def _hello_name(name: str):
    print(f"Hello, {name!s}!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Both `style` and `selected_style` can be lazily evaluated if necessary. Simply pass a callable with a signature similar to the following:

```
from action_completer.types import ActionCompletable_T

def dynamic_style(completable: ActionCompletable_T, completable_value: str) -> str:
    ...
```

For example, let's say we want to style the "John" completion with a blue background and leave everyone else with a red background:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter, types

completer = ActionCompleter()


def dynamic_style(
    completable: types.ActionCompletable_T, completable_value: str
) -> str:
    if completable_value.lower() == "john":
        return "fg:white bg:blue"

    return "fg:white bg:red"


@completer.action("hello")
@completer.param(
    ["Mark", "John", "William"],
    style=dynamic_style,
    selected_style="bold",
)
def _hello_name(name: str):
    print(f"Hello, {name!s}!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Warning: Take note of the return type for the dynamic style value. This callable has the constraint that it **requires** we always return a string of some kind; it should never be allowed to return `None`. To omit from styling a completion, simply return an empty string. Typically it is safest to return early for any desired stylings you would like to apply and leave an empty string as the last available return value at the bottom of the callable:

```
def dynamic_style(completable: ActionCompletable_T, completable_value: str) -> str:
    # logic to determine the desired style for a completion result
    ...

    # always return AT LEAST an empty string
    return ""
```

Text

You can customize the actual display of the completion text itself by passing some string to the `display` keyword argument:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()
```

(continues on next page)

(continued from previous page)

```
@completer.action("hello", display="Run hello world")
def _hello_world():
    print("Hello, World!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

We can also pass an instance of `FormattedText` to `display` to get some pretty fancy completions:

```
from prompt_toolkit.formatted_text import HTML, to_formatted_text
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello", display=to_formatted_text(HTML("Run <i>hello world</i>")))
def _hello_world():
    print("Hello, World!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Similar to the `style` and `selected_style` properties, we can define the `display` lazily through a callable. The callable for the `display` should have a signature similar to the following:

```
from typing import Union

from prompt_toolkit.formatted_text import FormattedText
from action_completer.types import ActionCompletable_T

def dynamic_display(
   completable: ActionCompletable_T, completable_value: str
) -> Union[str, FormattedText]:
   ...
```

You can also easily include the completion value within a string or `FormattedText` instance without having to do anything fancy by supplying a `{completion}` format in the `display` or `display_meta` values. For example:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()
```

(continues on next page)

(continued from previous page)

```
@completer.action("hello")
@completer.param(["1", "2", "3"], cast=int, display_meta="Says hello to {completion}")
def _hello_action(num: int):
    print(f"Hello, {num!s}!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

This will produce completions where the completed value is inserted into the template provided for the `display_meta`.

Somewhat useful if you don't want to go through the effort of defining a callable to just place completion values in your completion's display or descriptions.

Description

If you would like to add a little helpful description to a completion, you can do so through the `display_meta` keyword argument.

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


@completer.action("hello", display_meta="Run hello world")
def _hello_world():
    print("Hello, World!")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
completer.run_action(prompt_result)
```

Everything that you can do with `display`, you can also do with `display_meta`. For more details about what kind of formats you can display, you should read through the prompt-toolkit documentation

1.2.7 Conditional Actions

Both *`Action`* and *`ActionGroup`* can be gated behind a `Filter` to indicate if the action or group should be considered completable. This filter is provided through the `active` keyword argument; by default it is set to `None`. If this filter is provided, it will be evaluated during completion and, if it evaluates to a falsy value, completion will not occur for the associated action or group.

For example, here is a quick (and dirty) method for only allowing a `hello` action to run once the user has first called `activate`:

```
from prompt_toolkit.filters import Condition
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()


ACTIVE: bool = False


def is_active() -> bool:
    return ACTIVE


def set_active(state: bool) -> bool:
    global ACTIVE
    ACTIVE = state
    return ACTIVE


@completer.action("activate", active=Condition(lambda: not is_active()))
def _activate_action():
    set_active(True)


@completer.action("deactivate", active=Condition(is_active))
def _deactivate_action():
    set_active(False)


@completer.action("hello", active=Condition(is_active))
def _hello_world():
    print("Hello, World!")


while True:
    prompt_result = prompt(
        ">>> ", completer=completer, validator=completer.get_validator()
    )
    completer.run_action(prompt_result)
```

1.2.8 Custom Action Execution

If you ever need to tweak how the requested action is executed, you can fetch a `partial()` instance from the completer instead of just calling *`run_action()`*. Use the *`get_partial_action()`* method with the output of the prompt to get an callable for the desired action with the casted parameters applied:

```
from prompt_toolkit.shortcuts import prompt

from action_completer import ActionCompleter

completer = ActionCompleter()
```

(continues on next page)

(continued from previous page)

```
@completer.action("hello")
@completer.param(["World", "Stephen"])
def _hello_action(name: str, *args):
    print(f"Hello, {name!s}!")
    print(f"Additional: {args!s}")


prompt_result = prompt(">>> ", completer=completer, validator=completer.get_
↪validator())
action_partial = completer.get_partial_action(prompt_result)

print(action_partial)
action_partial("I'm something new")
```

There you have it.

That is a basic overview of the features currently available in the *`ActionCompleter`*. To understand more about what is available to help to construct basic autocompletion for prompt interfaces, I would recommend you read through the prompt-toolkit documentation and continue on reading through the *Action Completer Package* internal reference docs.

CHAPTER

TWO

CONTRIBUTING

Important: When contributing to this repository, please adhere to our code-of-conduct and first discuss the change you wish to make via an issue **before** submitting a pull request.

2.1 Local Development

The following sections will guide you through setting up a local development environment for working on this project package. **At the very least**, make sure that you have the necessary pre-commit hooks installed to make sure that all commits are pristine before they make it into the change history.

2.1.1 Installing Python

Note: If you already have Python 3.7+ installed on your local system, you can skip this step completely.

Installing Python should be done through pyenv. To first install `pyenv` please follow the guide they provided at https://github.com/pyenv/pyenv#installation. When you finally have `pyenv` you should be good to continue on.

```
$ pyenv --version
pyenv x.x.x
```

Now that you have `pyenv` we can install the necessary Python version. This project's package depends on Python 3.7+, so we can request that through `pyenv`.

```
$ pyenv install 3.7  # to install Python 3.7+
...

$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name
$ pyenv local 3.7  # to mark the project directory as needing Python 3.7+
...

$ pyenv global 3.7  # if you wish Python 3.7 to be aliased to `python` everywhere
...
```

After installing and marking the repository as requiring Python 3.7+ you should be good to continue on installing the project's dependencies.

2.1.2 Virtual Environment

We use Poetry to manage both our dependencies and virtual environments. Setting up `poetry` just involves installing it through `pip` as a user-level dependency.

```
$ pip install --user poetry
Collecting poetry
Downloading poetry-x.x.x-py2.py3-non-any.whl
...
```

You can quickly setup your entire development environment by running the installation process from poetry.

```
$ poetry install
Installing dependencies from lock file
...
```

This with create a virtual environment for you and install the necesary development dependencies. From there you can jump into a subshell using the newly created virtual environment using the `shell` subcommand.

```
$ poetry shell
pawning shell within ~/.local/share/virtualenvs/my-project-py3.7
...

$ exit # when you wish to exit the subshell
```

From this shell you have access to all the necessary development dependencies installed in the virutal environment and can start actually writing and running code within the client package.

2.1.3 Style Enforcement

This project's preferred styles are fully enforced through pre-commit hooks. In order to take advantage of these hooks please make sure that you have `pre-commit` and the configured hooks installed in your local environment.

Installing `pre-commit` is done through `pip` and should be installed as a user-level dependency as it adds some console scripts that all projects using `pre-commit` will need.

```
$ pip install --user pre-commit
Collecting pre-commit
Downloading pre_commit-x.x.x-py2.py3-none-any.whl
...

$ pre-commit --version
pre-commit 2.4.0
```

Once `pre-commit` is installed you should also install the hooks into the cloned repository.

```
$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name

$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

After this you should be good to continue on. These installed hooks will do a first-time setup when you attempt your next commit to build hook environments. Changes that violate the defined style specifications in `setup.cfg` and `pyproject.toml` will cause the commit to fail and will likely make the necessary changes to added / changed files to be written to the failing files.

This will give you the opprotunity to view the changes the hooks made to the failing files and add the new changes to the commit in order to make the commit pass. It also gives you the opprotunity to make tweaks to the autogenerated changes to make them more human accessible (only if necessary).

2.1.4 Editor Configuration

We also have some specific settings for editor configuration via editorconfig. We recommend you install the appropriate plugin for your editor of choice if your editor doesn't already natively support `.editorconfig` configuration files.

2.1.5 Project Tasking

All of our tasks are built and run through invoke which is basically just a more advanced (a little too advanced) Python alternative to make. The only reason we are using this utility is because I know how it works and I already had most of the necessary tasks defined from other projects.

From within the Poetry subshell, you can access and run these commands through the provided `invoke` development dependency.

```
$ invoke --list
Available tasks:

   build                  Build the project.
   clean                  Clean the project.
   lint                   Lint the project.
   profile                Run and profile a given Python script.
   test                   Test the project.
   docs.build             Build docs.
   docs.build-news        Build towncrier newsfragments.
   docs.clean             Clean built docs.
   docs.view              Build and view docs.
   linter.black           Run Black tool check against source.
   linter.flake8          Run Flake8 tool against source.
   linter.isort           Run ISort tool check against source.
   linter.mypy            Run MyPy tool check against source.
   package.build          Build pacakge source files.
   package.check          Check built package is valid.
   package.clean          Clean previously built package artifacts.
   package.format         Auto format package source files.
   package.requirements   Generate requirements.txt from Poetry's lock.
   package.stub           Generate typing stubs for the package.
   package.test           Run package tests.
   package.typecheck      Run type checking with generated package stubs.
```

You can run these tasks to do many miscellaneous project tasks such as building documentation.

```
$ invoke docs.build
[docs.build] ... building 'html' documentation
Running Sphinx v3.0.3
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
no targets are out of date.
```

(continues on next page)

(continued from previous page)

```
build succeeded.

The HTML pages are in build/html.
```

All of these tasks should just work right out of the box, but something might break eventually after required tooling gets enough major updates.

2.2 Opening Issues

Issues should follow the included `ISSUE_TEMPLATE` found in `.github/ISSUE_TEMPLATE.md`.

- **Issues should contain the following sections:**
  - Expected Behavior
  - Current Behavior
  - Possible Solution
  - Steps to Reproduce (for bugs)
  - Context
  - Your Environment

These sections help the developers greatly by providing a large understanding of the context of the bug or requested feature without having to launch a full fleged discussion inside of the issue.

2.3 Creating Pull Requests

Pull requests should follow the included `PULL_REQUEST_TEMPLATE` found in `.github/PULL_REQUEST_TEMPLATE.md`.

- Pull requests should always be from a `topic` / `feature` / `bugfix` (left side) branch.
  Pull requests from master branches will not be merged.
- Pull requests should not fail our requested style guidelines or linting checks.

CHAPTER

THREE

CHANGELOG

All notable changes to this project will be documented in this file.
The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

3.1 1.1.1 (*2020-09-24*)

3.1.1 Bug Fixes

- Declaring dependency on `typing-extensions` in `pyproject.toml`. #5

3.2 1.1.0 (*2020-09-24*)

3.2.1 Features

- Adding support for formatting `{completion}` in LazyText_T values for completions. #4

3.2.2 Bug Fixes

- Ensuring that prompt text does not reduce to nothing before attempting to fuzzy match against explicit completions and validation. #3

3.2.3 Miscellaneous

- Moving `ActionValidator._get_best_choice` logic into `utils.get_best_choice`.

3.3 1.0.0 (*2020-09-09*)

3.3.1 Miscellaneous

- Adding the basic first release of the action completer that I have used in the past for personal projects. Note that this is currently untested and I have since lost the git history for this project (I know, I know...).
- Creating full test-suite for most basic (non-edge case) functionality.

CHAPTER

FOUR

LICENSE

```
ISC License

Copyright (c) 2020, Stephen Bunn <stephen@bunn.io>

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

CHAPTER

FIVE

PROJECT REFERENCE

5.1 Action Completer Package

Below is autogenerated documentation directly from docstrings in the source.
If you are just getting started, I would recommend you first read the *Getting Started* documentation. The package documentation included here should mostly just be used for reference or a slightly more simple look into the internals of the project.

5.1.1 Types

Contains the base data types used to power the completion and validation of actions.

`action_completer.types.`**`ActionCompletable_T`**
Defines the base data types that are typically used for completion. This type feels like it should be used more, but it really isn't necessary that often.

Type typing.Type

`action_completer.types.`**`ActionContext_T`**
Defines the value types to expect from the extraction of the context for the provided prompt document. Since we need various details about the current state of the prompt buffer, we explicitly define the types of the extracted context tuple through this type.

Type typing.Type

`action_completer.types.`**`ActionParamBasic_T`**
Defines the allowable types for basic completion and validation. Typically just a single value, in this case a string.

Type typing.Type

`action_completer.types.`**`ActionParamIterable_T`**
Defines the allowable types for iterable completion and validation. Only supports hashable iterables (tuples, lists) of strings.

Type typing.Type

`action_completer.types.`**`ActionParamCompleter_T`**
Defines the allowable types for nested completer completion (not validation). Currently just an alias of `Completer`.

Type typing.Type

action_completer.types.**ActionParamCallable_T**
Defines the allowable type for callable completions and validation.

Type typing.Type

action_completer.types.**ActionParamSource_T**
Defines the allowable types for action parameter sources that can be completed and validated. This is a union of previously defined action param types.

Type typing.Type

action_completer.types.**ActionParamValidator_T**
Defines the allowable types for validation callables or instances.

Type typing.Type

action_completer.types.**LazyString_T**
Defines the allowable types for optionally lazy evaluated strings. This is either just an instance of a string or a callable that results in a string.

Type typing.Type

action_completer.types.**LazyText_T**
Similar to *LazyString_T*, this defines the allowable types for optionally lazy evaluated strings or instances of FormattedText. This is either just an instance or a callable that results in an instance.

Type typing.Type

class action_completer.types.**Action**(*action=None*, *params=None*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *active=None*, *capture_all=False*)
Defines a completable action.

action
The callable function that should be completeable and can be called after validation of the given action and associated parameters

Type Optional[Callable[.., Any]], optional

params
The list of action parameters in the same order of positional arguments for the action callable.

Type Optional[List[*ActionParam*]], optional

style
The style string to apply to completion results for the action

Type Optional[*LazyString_T*], optional

selected_style
The style string to apply to selected completion results for the action

Type Optional[*LazyString_T*], optional

display
The custom display to apply to completion results for the action

Type Optional[*LazyText_T*], optional

display_meta
The custom display meta (description) to apply to completion results for the action

Type Optional[*LazyText_T*], optional

active
A callable filter that results in a boolean to indicate if the action should be considred as active and displayed as a completion result

Type Optional[Filter], optional

capture_all
If there is the option for this action to accept more arguments than defined by the provided parameters, this flag will allow for any number of following arguments (after parameters) as additional positional arguments to the provided `action` callable, defaults to False

Type bool, optional

class action_completer.types.**ActionGroup**(*children*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *active=None*)
Defines a completable group of either nested action groups or leaf actions.

Important: It is crucial for proper fragment extraction that no children keys contain spaces. Each non-escaped space is considered a new fragment and used to find the context for both completion and validation. Although it would technically be possible to support child key completion with spaces, it involves more complex features of prompt-toolkit than *just* completion.

Therefore, I have decided to hold the opinion that no spaces should be allowable as keys in a group's children. A post-initialization validator exists to ensure this is true for any action group you define.

---

children
The dictionary of completion text (keys) to a nested action group or a callable action (value) that forms the group's children

Type Dict[str, Union[*ActionGroup*, *Action*]]

style
The style string to apply to completion results for the action group

Type Optional[*LazyString_T*], optional

selected_style
The style string to apply to selected completion results for the action group

Type Optional[*LazyString_T*], optional

display
The custom display to apply to completion results for the action group

Type Optional[*LazyText_T*], optional

display_meta
The custom display meta (description) to apply to completion results for the action group

Type Optional[*LazyText_T*], optional

active
A callable filter that results in a boolean to indicate if the action group should be considred as active and displayed as a completion result

Type Optional[Filter], optional

action(*name*, *params=None*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *active=None*, *capture_all=False*)
Decorate a callable as an action within the current group.

Basic root level actions are easily defined directly off of the completer instance like follows:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

@completer.action("hello")
def _hello_action():
    print("Hello, World!")

completer.run_action(prompt(">>> ", completer=completer))

# available through the following prompt:
# >>> hello
# Hello, World!
```

You can nest actions in sub-groups by first calling *group()* to define a new group and base all actions from the `action` decorator provided on that new group.

Parameters

- **name** (*str*) – The completion name that triggers this action
- **params** (*Optional[List[ActionParam]], optional*) – The list of action parameters to complete and handle within the action
- **style** (Optional[*LazyString_T*], optional) – The style string to apply to completion results for the action
- **selected_style** (Optional[*LazyString_T*], optional) – The style string to apply to selected completion results for the action
- **display** (Optional[*LazyText_T*], optional) – The custom display to apply to completion results for the action
- **display_meta** (Optional[*LazyText_T*], optional) – The custom display meta (description) to apply to completion results for the action
- **active** (*Optional[Filter], optional*) – A callable filter that results in a boolean to indicate if the action group should be considred as active and displayed as a completion result
- **capture_all** (*bool, optional*) – If there is the option for this action to accept more arguments than defined by the provided parameters, this flag will allow for any number of following arguments (after parameters) as additional positional arguments to the provided `action` callable, defaults to False

Raises

- **ValueError** – When the given action name contains spaces
- **ValueError** – When the given action name is already present in the group

Returns The newly wrapped action callable

Return type Callable[.., Any]

group(*name*, *children=None*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *active=None*)

Create a new subgroup for the current group.

By default the *ActionCompleter* comes with a `root` group to extend from. However, if you want to build a set of nested commands, you can use this function to register a new group on the completer.

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

nested_group = completer.group("hello")

@nested_group.action("world")
def _hello_world_action():
    print("Hello, World!")


completer.run_action(prompt(">>> ", completer=completer))
# available through the following prompt:
# >>> hello world
# Hello, World!
```

Parameters

- **name** (*str*) – The completion text that triggers this group
- **children** (*Dict[str, Union[ActionGroup, Action]]*) – The dictionary of completion text (keys) to a nested action group or a callable action (value) that forms the group's children
- **style** (Optional[*LazyString_T*], optional) – The style string to apply to completion results for the action group
- **selected_style** (Optional[*LazyString_T*], optional) – The style string to apply to selected completion results for the action group
- **display** (Optional[*LazyText_T*], optional) – The custom display to apply to completion results for the action group
- **display_meta** (Optional[*LazyText_T*], optional) – The custom display meta (description) to apply to completion results for the action group
- **active** (*Optional[Filter], optional*) – A callable filter that results in a boolean to indicate if the action group should be considered as active and displayed as a completion result

Raises

- **ValueError** – When the provided group name contains spaces
- **ValueError** – When the provided group name is already in the current group

Returns The created action group

Return type *ActionGroup*

class action_completer.types.**ActionParam**(*source*, *cast=None*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *validators=None*)

Defines a completable action parameter.

source
The completion source for the parameter

Type *ActionParamSource_T*

cast
The type to cast the action parameter to during execution of the action tied to this parameter

Type Optional[Type], optional

style
The style string to apply to completion results for the action parameter

Type Optional[*LazyString_T*], optional

selected_style
The style string to apply to selected completion results for the action parameter

Type Optional[*LazyString_T*], optional

display
The custom display to apply to completion results for the action parameter

Type Optional[*LazyText_T*], optional

display_meta
The custom display meta (description) to apply to completion results for the action parameter

Type Optional[*LazyText_T*], optional

validators
The list of validators to run in-order against the parameter value during validation

Type Optional[List[*ActionParamValidator_T*]], optional

action_completer.types.**param**(*source*, *cast=None*, *style=None*, *selected_style=None*, *display=None*, *display_meta=None*, *validators=None*)

Create a new action parameter for an action.

Basic parameters can be defined right before defining a method as an action.

```
from pathlib import Path
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.validation import Validator
from action_completer import ActionCompleter
completer = ActionCompleter()

@completer.action("cat")
@completer.param(
    PathCompleter(),
    cast=Path,
    validators=[Validator.from_callable(lambda p: Path(p).is_file())]
)
def _cat_action(filepath: Path):
    with filepath.open("r") as file_handle:
        print(file_handle.read())

completer.run_action(prompt(">>> ", completer=completer))
# available through the following prompt:
# # $ echo "my content" > ./my-file.txt
# >>> cat ./my-file.txt
# my content
```

Important: The application order of `param` decorators is very important. Since these defined action parameters are applied as positional arguments to the action, we are opinionated on the ordering that decorators should be applied.

We purposefully reverse the traditional decorator application order to make the readability of actions created purely through decorators a bit easier. This means that the **last** parameter should be the **first** decorator applied to the action callable, and the **very last** decorator applied to the action callable should be the action decorator.

This benefits readability by ordering the defined param decorators top-to-bottom as arguments applied left-to-right in the action callable.

```
# valid
@action("test")
@param("source1")
@param("source2")
def _valid_action(source1: str, source2: str):
    ...

# invalid
@param("source2")
@param("source1")
@action("test")
def _invalid_action(source1: str, source2: str):
    ...

# also invalid
@action("test")
@param("source2")
@param("source1")
def _also_invalid_action(source1: str, source2: str):
    ...
```

We have *some* checks to raise warnings in case you accidentally use an invalid spec for applying the decorators. Although I'm not 100% sure it will capture all the possible states of defining actions that you might come up with.

If you don't like this design, I would recommend you instead pass *ActionParam* instances to the `params` keyword argument when using the *action()* decorator.

Parameters

- **source** (*ActionParamSource_T*) – The completion source for the parameter
- **cast** (*Optional[Type], optional*) – The type to cast the action parameter to during execution of the action tied to this parameter
- **style** (Optional[*LazyString_T*], optional) – The style string to apply to completion results for the action parameter
- **selected_style** (Optional[*LazyString_T*], optional) – The style string to apply to selected completion results for the action parameter
- **display** (Optional[*LazyText_T*], optional) – The custom display to apply to completion results for the action parameter
- **display_meta** (Optional[*LazyText_T*], optional) – The custom display meta (description) to apply to completion results for the action parameter
- **validators** (Optional[List[*ActionParamValidator_T*]], optional) – The list of validators to run in-order against the parameter value during validation

Returns The newly wrapped action with the defined parameters

Return type Callable[.., Any]

5.1.2 Completer

Contains the completer for a single root *ActionGroup* instance.

Registering actions for the completer can be done in several different ways. The simplest way, is to use the provided *action()* decorator:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

@completer.action("hello")
def _hello_world():
    print("Hello, World!")

completer.run_action(prompt(">>> ", completer=completer))
```

Another common way is to build the root *ActionGroup* structure yourself:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter, ActionGroup, Action

def _hello_world():
    print("Hello, World!")

root_group = ActionGroup({"hello": Action(_hello_world)})
completer = ActionCompleter(root_group)

completer.run_action(prompt(">>> ", completer=completer))
```

Lastly, if you *really* need it, you can specify the starting structure of the completer through a **very specific** dictionary structure. I personally needed this in the past and haven't yet had the chance to remove the need of it.

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter

def _hello_world():
    print("Hello, World!")

completer = ActionCompleter.from_dict({
    "root": {
        "hello": {
            "action": _hello_world
        }
    },
    "fuzzy_tolerance: 75
})

completer.run_action(prompt(">>> ", completer=completer))
```

class action_completer.completer.**ActionCompleter**(*root=None*, *fuzzy_tolerance=75*)

Custom completer for actions.

This completer should be relatively easy to consume and provides features for extending completions with custom styles and display properties.

The most straight forward method of defining actions is to use the *action()* decorator right off of the completer:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

@completer.action("hello")
def _hello_world():
    print("Hello, World!")

output = prompt(">>> ", completer=completer)
completer.run_action(output)

# >>> hello
# Hello, World!
```

If you want to go a more declarative route, you can explicitly build the tree structure of *ActionGroup*, *Action*, and *ActionParam* yourself.

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompelter, ActionGroup, Action, ActionParam

def _hello_world():
    print("Hello, World!")

ACTIONS = ActionGroup({
    "hello": Action(_hello_world)
})

completer = ActionCompleter(ACTIONS)

output = prompt(">>> ", completer=completer)
completer.run_action(output)

# >>> hello
# Hello, World!
```

get_completions(*document*, *complete_event*)
Generate completions for the given prompt document.

Parameters

- **document** (*Document*) – The document directly from the prompt to generate completions for
- **complete_event** (*CompleteEvent*) – The completion event for the completions

Yields *prompt_toolkit.completion.Completion* – A completion for the given prompt document

Return type Generator[Completion, None, None]

get_partial_action(*prompt_result*)
Get the partial for the action callable with action parameters included.

Parameters **prompt_result** (*str*) – The result of the completer's prompt call

Raises **ValueError** – When no actionable action can be determined from the given prompt result

Returns A partial callable for the related action including clean parameters

Return type Callable[.., Any]

get_validator()
Get an instance of the validator for the current completer.

Returns A prompt validator for the current state of the completer

Return type *ActionValidator*

run_action(*prompt_result*, **args*, ***kwargs*)
Run the related action from the given prompt result.

Parameters **prompt_result** (*str*) – The result of the completer's prompt call

Returns Whatever the return value of the related action callable is

Return type Any

class action_completer.completer.**ActionParamCompletionIterator_T**(**args*, ***kwds*)
Protocol typing for param-level Completion generators.

Required for dynamic switching between various action param completion interators.

class action_completer.completer.**CompletionIterator_T**(**args*, ***kwds*)
Protocol typing for top-level (non-param) Completion generators.

Required for dynamic switching between group and action completion iterators since both ActionGroup and Action can appear as values in ActionGroup.children.

5.1.3 Validator

Contains the validator used to validate data produced by the action completer.

To retreive this validator, it is highly recommend that you utilize the *get_validator()* method. If your completer instance is dynamic, you probably want to fetch this validator instance for every call to prompt().

For example:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter
completer = ActionCompleter()

# ... register completer actions

while True:
    # note the call to `get_validator` for every call to prompt
    prompt_result = prompt(
        ">>> ",
        completer=completer,
        validator=completer.get_validator()
    )
```

You could also initialize the *ActionValidator* yourself by passing through the root group of the *ActionCompleter*:

```
from prompt_toolkit.shortcuts import prompt
from action_completer import ActionCompleter, ActionValidator
completer = ActionCompleter()

# ... register completer actions

while True:
```

(continues on next page)

(continued from previous page)

```
    validator = ActionValidator(completer.root)
    prompt_result = prompt(
        ">>> ",
        completer=completer,
        validator=validator
    )
```

class action_completer.validator.**ActionValidator**(*root*)

Custom validator for a *ActionCompleter*.

Most of the time you should get this instance from *get_validator()* however if you need to build an instance of it yourself you can use the following logic:

```
from action_completer.completer import ActionCompleter
from action_completer.validator import ActionValidator

completer = ActionCompleter()
validator = ActionValidator(completer.root)
```

validate(*document*)

Validate the current document from the *ActionCompleter*.

Parameters **document** (*Document*) – The document to validate

Raises **ValidationError** – When validation of the given document fails.

5.1.4 Utilities

Contains utility functions used throughout various points of the module.

action_completer.utils.**decode_completion**(*text*)

Reverse the encoding process for completion text.

Parameters **text** (*str*) – The text to decode for use in action parameters and displaying

Returns The properly decoded text

Return type str

action_completer.utils.**encode_completion**(*text*)

Encode some completion text for writing to the user's current prompt buffer.

Parameters **text** (*str*) – The text to encode for writing

Returns The properly encoded text for the user's prompt buffer

Return type str

action_completer.utils.**extract_context**(*action_group*, *fragments*)

Extract the current context for a root action group and buffer fragments.

Parameters

- **action_group** (ActionGroup) – The root action group to start context extraction
- **fragments** (*List[str]*) – The text fragments extracted from the current user's prompt buffer

Returns A tuple of (parent ActionGroup, parent name, current ActionGroup/Action, list of remaining fragments [parameters])

Return type *ActionContext_T*

action_completer.utils.**format_dynamic_value**(*template*, *text*)
Format the given template text for the dynamic value.

Parameters

- **template** (*str*) – The template text to be formatted
- **text** (*str*) – The current text fragment that triggered the completion

Returns The formatted text

Return type str

action_completer.utils.**get_best_choice**(*choices*, *user_value*)
Guess the best choice from an interable of choice strings given a target value.

This method has a few caveats to make using it with completion a bit easier:

- If no choices are given, nothing is returned.
- If only 1 choice is given, that choice is always returned.
- If the given value (taget) text is not alphanumerical, the first available choice is returned.

Parameters

- **choices** (*Iterable[str]*) – The iterable of choices to guess from
- **user_value** (*str*) – The target value to base the best guess off of

Returns The best choice if available, otherwise None

Return type Optional[str]

action_completer.utils.**get_dynamic_value**(*source*, *value*, *text*, *default=None*)
Resolve a lazy/dynamic completion format value.

The given value will be formatted in place of any `{completion}` usage within the dynamic text. The following example will display the description containing the completion value in place of the given value

```
@completer.action("hello-world")
@completer.param(["1", "2", "3"], display_meta="Will display {completion}")
def _hello_world(number_value: str):
    print(f"Hello, {number_value!s}!")
```

Parameters

- **source** (*ActionCompletable_T*) – The source for the completion
- **value** (*LazyText_T*) – The dynamic value that needs to be resolved for the source
- **text** (*str*) – The current text fragment that triggered the given source
- **default** (*Optional[Union[str, FormattedText]]*) – A default if the given value resolves to None. Defaults to None.

Returns Either a string or `FormattedText` instance if the value is properly resolved, otherwise defaults to the default

Return type Optional[Union[str, FormattedText]]

action_completer.utils.**get_fragments**(*text*)
Get the properly split fragments from the current user's prompt buffer.

Parameters **text** (*str*) – The text of the current user's prompt buffer

Returns A list of string fragments

Return type List[str]

action_completer.utils.**iter_best_choices**(*choices*, *user_value*, *fuzzy_tolerance=None*)

Iterate over the sorted closest strings from some choices using fuzzy matching.

This iterator has a few caveats that make using it with completion a bit easier:

- If no choices are given, nothing is ever yielded.
- If only 1 choice is given, that choice will always be yielded.

  Basically no fuzzy matching will occur against to allow for filtering out just a single choice.
- If the given value (target) text is empty, all choices will be yielded.
- If the given value (target) text is not alphanumerical, all choices are yielded.

Parameters

- **choices** (*Iterable[str]*) – An interable of strings to apply fuzzy matching to
- **user_value** (*str*) – The value to use for fuzzy comparison against choices
- **fuzzy_tolerance** (*Optional[int], optional*) – The percentage integer 0-100 to tolerate the resulting fuzzy matches. Defaults to None.

Yields *str* – Sorted best matching strings from choices in comparison to `user_value`

Return type `Generator`[`str`, `None`, `None`]

action_completer.utils.**noop**(**args*, ***kwargs*)

Noop function that does absolutely nothing.

Return type `None`

PYTHON MODULE INDEX

a

INDEX