argcomplete Documentation

Andrey Kislyuk

May 08, 2018

Contents

Tab complete all the things!

Argcomplete provides easy, extensible command line tab completion of arguments for your Python script.

It makes two assumptions:

- You're using bash as your shell (limited support for zsh and tcsh is available)
- You're using argparse to manage your command line arguments/options

Argcomplete is particularly useful if your program has lots of options or subparsers, and if your program can dynamically suggest completions for your argument/option values (for example, if the user is browsing resources over the network).

CHAPTER 1

Installation

```
pip install argcomplete
activate-global-python-argcomplete
```

See *Activating global completion* below for details about the second step (or if it reports an error).

Refresh your bash environment (start a new shell or `source /etc/profile`).

CHAPTER 2

Synopsis

Python code (e.g. `my-awesome-script`):

```
#!/usr/bin/env python
# PYTHON_ARGCOMPLETE_OK
import argcomplete, argparse
parser = argparse.ArgumentParser()
...
argcomplete.autocomplete(parser)
args = parser.parse_args()
...
```

Shellcode (only necessary if global completion is not activated - see *Global completion* below), to be put in e.g. `.bashrc`:

```
eval "$(register-python-argcomplete my-awesome-script)"
```

2.1 argcomplete.autocomplete(*parser*)

This method is the entry point to the module. It must be called **after** ArgumentParser construction is complete, but **before** the `ArgumentParser.parse_args()` method is called. The method looks for an environment variable that the completion hook shellcode sets, and if it's there, collects completions, prints them to the output stream (fd 8 by default), and exits. Otherwise, it returns to the caller immediately.

Side effects

Argcomplete gets completions by running your program. It intercepts the execution flow at the moment `argcomplete.autocomplete()` is called. After sending completions, it exits using `exit_method` (`os._exit` by default). This means if your program has any side effects that happen before `argcomplete` is called, those side effects will happen every time the user presses `<TAB>` (although anything your program prints to stdout

or stderr will be suppressed). For this reason it's best to construct the argument parser and call `argcomplete.autocomplete()` as early as possible in your execution flow.

Performance

If the program takes a long time to get to the point where `argcomplete.autocomplete()` is called, the tab completion process will feel sluggish, and the user may lose confidence in it. So it's also important to minimize the startup time of the program up to that point (for example, by deferring initialization or importing of large modules until after parsing options).

CHAPTER 3

Specifying completers

You can specify custom completion functions for your options and arguments. Two styles are supported: callable and readline-style. Callable completers are simpler. They are called with the following keyword arguments:

- `prefix`: The prefix text of the last word before the cursor on the command line. For dynamic completers, this can be used to reduce the work required to generate possible completions.
- `action`: The `argparse.Action` instance that this completer was called for.
- `parser`: The `argparse.ArgumentParser` instance that the action was taken by.
- `parsed_args`: The result of argument parsing so far (the `argparse.Namespace` args object normally returned by `ArgumentParser.parse_args()`).

Completers should return their completions as a list of strings. An example completer for names of environment variables might look like this:

```
def EnvironCompleter(**kwargs):
    return os.environ
```

To specify a completer for an argument or option, set the `completer` attribute of its associated action. An easy way to do this at definition time is:

```
from argcomplete.completers import EnvironCompleter

parser = argparse.ArgumentParser()
parser.add_argument("--env-var1").completer = EnvironCompleter
parser.add_argument("--env-var2").completer = EnvironCompleter
argcomplete.autocomplete(parser)
```

If you specify the `choices` keyword for an argparse option or argument (and don't specify a completer), it will be used for completions.

A completer that is initialized with a set of all possible choices of values for its action might look like this:

```
class ChoicesCompleter(object):
    def __init__(self, choices):
```

(continues on next page)

(continued from previous page)

```
        self.choices = choices

    def __call__(self, **kwargs):
        return self.choices
```

The following two ways to specify a static set of choices are equivalent for completion purposes:

```
from argcomplete.completers import ChoicesCompleter

parser.add_argument("--protocol", choices=('http', 'https', 'ssh', 'rsync', 'wss'))
parser.add_argument("--proto").completer=ChoicesCompleter(('http', 'https', 'ssh',
↪'rsync', 'wss'))
```

Note that if you use the `choices=<completions>` option, argparse will show all these choices in the `--help` output by default. To prevent this, set `metavar` (like `parser.add_argument("--protocol", metavar="PROTOCOL", choices=('http', 'https', 'ssh', 'rsync', 'wss'))`).

The following script uses `parsed_args` and Requests to query GitHub for publicly known members of an organization and complete their names, then prints the member description:

```
#!/usr/bin/env python
# PYTHON_ARGCOMPLETE_OK
import argcomplete, argparse, requests, pprint

def github_org_members(prefix, parsed_args, **kwargs):
    resource = "https://api.github.com/orgs/{org}/members".format(org=parsed_args.
↪organization)
    return (member['login'] for member in requests.get(resource).json() if member[
↪'login'].startswith(prefix))

parser = argparse.ArgumentParser()
parser.add_argument("--organization", help="GitHub organization")
parser.add_argument("--member", help="GitHub member").completer = github_org_members

argcomplete.autocomplete(parser)
args = parser.parse_args()

pprint.pprint(requests.get("https://api.github.com/users/{m}".format(m=args.member)).
↪json())
```

Try it like this:

```
./describe_github_user.py --organization heroku --member <TAB>
```

If you have a useful completer to add to the completer library, send a pull request!

3.1 Readline-style completers

The readline module defines a completer protocol in **rlcompleter_**. Readline-style completers are also supported by argcomplete, so you can use the same completer object both in an interactive readline-powered shell and on the bash command line. For example, you can use the readline-style completer provided by IPython to get introspective completions like you would get in the IPython shell:

```
import IPython
parser.add_argument("--python-name").completer = IPython.core.completer.Completer()
```

You can also use argcomplete.CompletionFinder.rl_complete to plug your entire argparse parser as a readline completer.

3.2 Printing warnings in completers

Normal stdout/stderr output is suspended when argcomplete runs. Sometimes, though, when the user presses <TAB>, it's appropriate to print information about why completions generation failed. To do this, use `warn`:

```
from argcomplete import warn

def AwesomeWebServiceCompleter(prefix, **kwargs):
    if login_failed:
        warn("Please log in to Awesome Web Service to use autocompletion")
    return completions
```

3.3 Using a custom completion validator

By default, argcomplete validates your completions by checking if they start with the prefix given to the completer. You can override this validation check by supplying the `validator` keyword to `argcomplete.autocomplete()`:

```
def my_validator(current_input, keyword_to_check_against):
    # Pass through ALL options even if they don't all start with 'current_input'
    return True

argcomplete.autocomplete(parser, validator=my_validator)
```

CHAPTER 4

Global completion

In global completion mode, you don't have to register each argcomplete-capable executable separately. Instead, bash will look for the string **PYTHON_ARGCOMPLETE_OK** in the first 1024 bytes of any executable that it's running completion for, and if it's found, follow the rest of the argcomplete protocol as described above.

Additionally, completion is activated for scripts run as `python <script>` and `python -m <module>`. This also works for alternate Python versions (e.g. `python3` and `pypy`), as long as that version of Python has argcomplete installed.

Bash version compatibility

Global completion requires bash support for `complete -D`, which was introduced in bash 4.2. On OS X or older Linux systems, you will need to update bash to use this feature. Check the version of the running copy of bash with `echo $BASH_VERSION`. On OS X, install bash via Homebrew (`brew install bash`), add `/usr/local/bin/bash` to `/etc/shells`, and run `chsh` to change your shell.

Global completion is not currently compatible with zsh.

Note: If you use setuptools/distribute `scripts` or `entry_points` directives to package your module, argcomplete will follow the wrapper scripts to their destination and look for `PYTHON_ARGCOMPLETE_OK` in the destination code.

If you choose not to use global completion, or ship a bash completion module that depends on argcomplete, you must register your script explicitly using `eval "$(register-python-argcomplete my-awesome-script)"`. Standard bash completion registration roules apply: namely, the script name is passed directly to `complete`, meaning it is only tab completed when invoked exactly as it was registered. In the above example, `my-awesome-script` must be on the path, and the user must be attempting to complete it by that name. The above line alone would **not** allow you to complete `./my-awesome-script`, or `/path/to/my-awesome-script`.

4.1 Activating global completion

The script `activate-global-python-argcomplete` will try to install the file `bash_completion.d/python-argcomplete.sh` (see on GitHub) into an appropriate location on your system (`/etc/bash_completion.d/` or `~/.bash_completion.d/`). If it fails, but you know the correct location of your bash completion scripts directory, you can specify it with `--dest`:

```
activate-global-python-argcomplete --dest=/path/to/bash_completion.d
```

Otherwise, you can redirect its shellcode output into a file:

```
activate-global-python-argcomplete --dest=- > file
```

The file's contents should then be sourced in e.g. `~/.bashrc`.

CHAPTER 5

Tcsh Support

To activate completions for tcsh use:

```
eval `register-python-argcomplete --shell tcsh my-awesome-script`
```

The `python-argcomplete-tcsh` script provides completions for tcsh. The following is an example of the tcsh completion syntax for `my-awesome-script` emitted by `register-python-argcomplete`:

```
complete my-awesome-script 'p@*@`python-argcomplete-tcsh my-awesome-script`@'
```

CHAPTER 6

Python Support

Argcomplete requires Python 2.7 or 3.3+.

CHAPTER 7

Common Problems

If global completion is not completing your script, bash may have registered a default completion function:

```
$ complete | grep my-awesome-script
complete -F _minimal my-awesome-script
```

You can fix this by restarting your shell, or by running `complete -r my-awesome-script`.

CHAPTER 8

Debugging

Set the `_ARC_DEBUG` variable in your shell to enable verbose debug output every time argcomplete runs. This will disrupt the command line composition state of your terminal, but make it possible to see the internal state of the completer if it encounters problems.

CHAPTER 9

Acknowledgments

Inspired and informed by the optcomplete module by Martin Blais.

CHAPTER 10

Links

- Project home page (GitHub)
- Documentation (Read the Docs)
- Package distribution (PyPI)
- Change log

10.1 Bugs

Please report bugs, issues, feature requests, etc. on GitHub.

CHAPTER 11

License

Licensed under the terms of the Apache License, Version 2.0. 

CHAPTER 12

API documentation

argcomplete.**autocomplete**()
Use this to access argcomplete. See *argcomplete.CompletionFinder.__call__()*.

exception argcomplete.**ArgcompleteException**

class argcomplete.**CompletionFinder**(*argument_parser=None*, *always_complete_options=True*, *exclude=None*, *validator=None*, *print_suppressed=False*, *default_completer=<argcomplete.completers.FilesCompleter object>*, *append_space=None*)
Inherit from this class if you wish to override any of the stages below. Otherwise, use argcomplete.autocomplete() directly (it's a convenience instance of this class). It has the same signature as *CompletionFinder.__call__()*.

__call__(*argument_parser*, *always_complete_options=True*, *exit_method=<built-in function _exit>*, *output_stream=None*, *exclude=None*, *validator=None*, *print_suppressed=False*, *append_space=None*, *default_completer=<argcomplete.completers.FilesCompleter object>*)

Parameters

- **argument_parser** (argparse.ArgumentParser) – The argument parser to autocomplete on
- **always_complete_options** (*boolean or string*) – Controls the autocompletion of option strings if an option string opening character (normally -) has not been entered. If True (default), both short (-x) and long (--x) option strings will be suggested. If False, no option strings will be suggested. If long, long options and short options with no long variant will be suggested. If short, short options and long options with no short variant will be suggested.
- **exit_method** (*callable*) – Method used to stop the program after printing completions. Defaults to os._exit(). If you want to perform a normal exit that calls exit handlers, use sys.exit().
- **exclude** (*iterable*) – List of strings representing options to be omitted from autocompletion

- **validator** (*callable*) – Function to filter all completions through before returning (called with two string arguments, completion and prefix; return value is evaluated as a boolean)
- **print_suppressed** (*boolean*) – Whether or not to autocomplete options that have the `help=argparse.SUPPRESS` keyword argument set.
- **append_space** (*boolean*) – Whether to append a space to unique matches. The default is `True`.

Note: If you are not subclassing CompletionFinder to override its behaviors, use `argcomplete.autocomplete()` directly. It has the same signature as this method.

Produces tab completions for `argument_parser`. See module docs for more info.

Argcomplete only executes actions if their class is known not to have side effects. Custom action classes can be added to argcomplete.safe_actions, if their values are wanted in the `parsed_args` completer argument, or their execution is otherwise desirable.

__init__(*argument_parser=None*, *always_complete_options=True*, *exclude=None*, *validator=None*, *print_suppressed=False*, *default_completer=<argcomplete.completers.FilesCompleter object>*, *append_space=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

collect_completions(*active_parsers*, *parsed_args*, *cword_prefix*, *debug*)

Visits the active parsers and their actions, executes their completers or introspects them to collect their option strings. Returns the resulting completions as a list of strings.

This method is exposed for overriding in subclasses; there is no need to use it directly.

filter_completions(*completions*)

Ensures collected completions are Unicode text, de-duplicates them, and excludes those specified by `exclude`. Returns the filtered completions as an iterable.

This method is exposed for overriding in subclasses; there is no need to use it directly.

get_display_completions()

This function returns a mapping of option names to their help strings for displaying to the user

Usage:

```
def display_completions(substitution, matches, longest_match_length):
    _display_completions = argcomplete.autocomplete.get_display_completions()
    print("")
    if _display_completions:
        help_len = [len(x) for x in _display_completions.values() if x]

        if help_len:
            maxlen = max([len(x) for x in _display_completions])
            print("\n".join("{0:{2}} -- {1}".format(k, v, maxlen)
                             for k, v in sorted(_display_completions.items())))
        else:
            print("    ".join(k for k in sorted(_display_completions)))
    else:
        print(" ".join(x for x in sorted(matches)))

    import readline
    print("cli /> {0}".format(readline.get_line_buffer()), end="")
    readline.redisplay()
```

(continues on next page)

(continued from previous page)

```
...
readline.set_completion_display_matches_hook(display_completions)
```

quote_completions(*completions*, *cword_prequote*, *last_wordbreak_pos*)

If the word under the cursor started with a quote (as indicated by a nonempty `cword_prequote`), escapes occurrences of that quote character in the completions, and adds the quote to the beginning of each completion. Otherwise, escapes all characters that bash splits words on (`COMP_WORDBREAKS`), and removes portions of completions before the first colon if (`COMP_WORDBREAKS`) contains a colon.

If there is only one completion, and it doesn't end with a **continuation character** (`/`, `:`, or `=`), adds a space after the completion.

This method is exposed for overriding in subclasses; there is no need to use it directly.

rl_complete(*text*, *state*)

Alternate entry point for using the argcomplete completer in a readline-based REPL. See also rlcompleter. Usage:

```
import argcomplete, argparse, readline
parser = argparse.ArgumentParser()
...
completer = argcomplete.CompletionFinder(parser)
readline.set_completer_delims("")
readline.set_completer(completer.rl_complete)
readline.parse_and_bind("tab: complete")
result = input("prompt> ")
```

(Use `raw_input` instead of `input` on Python 2, or use eight).

`argcomplete.`**warn**(**args*)

Prints **args** to standard error when running completions. This will interrupt the user's command line interaction; use it to indicate an error condition that is preventing your completer from working.

CHAPTER 13

Table of Contents

Python Module Index

a

Index

Symbols

A

C

F

G

Q

R

W