Argvard Documentation

Release 0.3.1-dev

Daniel Neuhäuser

May 01, 2014

Welcome to Argvard's documentation. If you are new to Argvard, start with *Installation* and read the *Tutorial* afterwards.

CHAPTER 1

User's Guide

1.1 Installation

In order to get started you need Python 2.7, Python 3.3 or a higher Python 3 version. Argvard is tested on CPython and PyPy but using other interpreters should work just as well.

It is recommended that you install Argvard – or any other Python library – inside a virtual environment, created with virtualenv. This allows you to isolate the projects you are working on from each other and the system.

Within a virtual environment you can install Argvard using pip:

```
$ pip install argvard
```

This should take just a moment and then you are good to go.

1.2 Tutorial

This tutorial teaches you how to create command line applications using Argvard. We will explore the framework step-by-step using the example of a simple "Hello, World!" application, which we are going to over-engineer to explore the features provided by Argvard.

1.2.1 Creating a basic "Hello World!"-Application

The first step on our journey will be to create a simple "Hello World!" application. The first part of that application consists of importing what we need:

```
from argvard import Argvard
```

The `Argvard` object is central to every Argvard application and – for now – the only thing we are going to need. The next part of the application is creating such an object:

```
application = Argvard()
```

As you can see this is trivial as we do not have to pass it any arguments. As you can see from the name I gave it, for all intents and purposes it is the application. The next step is making that the application do something:

```
@application.main()
def main(context):
    print(u'Hello, World!')
```

The *application.main* decorator is used to register what in Argvard terms is called the *main* function. If you are familiar with other programming languages, you may be aware that a *main* function of some form can be found in many languages. In languages in which it exists it acts as an entry point and is automatically called when your application is started.

This case is similiar, *main* is a function that will always be called by the *application* after any options have been parsed. The *main* function is supposed to do, whatever your application is supposed to do.

The last step is calling the application:

```
if __name__ == '__main__':
   application()
```

If you are not already familiar with the pattern, *__name__* is a special variable the interpreter sets to the name of the current module. If the module is being executed directly (and is not just imported), *__name__* will be set to `'__main__'`. This ensures that *application* is not called, unless the module is executed directly, which makes it possible to import the module without any side effects.

Once you have typed that into your editor, save it as *hello.py* and execute it with `python hello.py`. It should print "Hello, World!" and exit.

Continue with *Dealing with positional arguments*.

1.2.2 Dealing with positional arguments

Now that we have managed to greet the world, let us be more specific about whom we greet or at least let users be able to do that:

```
@application.main('name')
def main(context, name):
    print(u'Hello, %s!' % name)
```

Replace the main function defined in *hello.py* with the code above. This defines a signature for the main function. A signature defines which positional arguments something takes, in this case a main function.

This signature defines one required positional argument called *name*. Positional arguments are passed to the function under the names defined in the signature.

Now in order to run *hello.py* you have to call it like this:

```
$ python hello.py Daniel
Hello, Daniel!
```

Obviously you can replace "Daniel" with your own name.

Default arguments

This does introduce a problem though because if we call run *hello.py* as we did previously, we get this:

```
$ python hello.py
error: name is missing
usage: hello.py [-h|--help] <name>
```

If we want to have backwards compatibility, we need to make the name an optional positional argument:

```
@application.main('[name]')
def main(context, name=u'World'):
    print(u'Hello, %s!' % name)
```

Now we can run the application with a name:

```
$ python hello.py Daniel
Hello, Daniel!
```

or without it:

```
$ python hello.py
Hello, World!
```

Repetitions

Now that we have managed to greet one person or well everyone. Let us try to greet multiple people:

```
@application.main('[name...]')
def main(context, name=None):
    if name is None:
        name = [u'World']
    if len(name) == 1:
        print(u'Hello, %s!' % name[0])
    elif len(name) == 2:
        print(u'Hello, %s and %s!' % (name[0], name[1]))
    else:
        print(u'Hello, %s and %s!' % (u', '.join(name[:-1]), name[-1]))
```

The function does quite a bit more than the previous ones, to achieve a nice formatting. Apart from that what has really changed is that we have added ... to the end of the argument in the signature.

Now we can greet any number of people:

```
$ python hello.py
Hello, World!
$ python hello.py Daniel
Hello, Daniel!
$ python hello.py Daniel Horst
Hello, Daniel and Horst!
$ python hello.py Daniel Horst Peter
Hello, Daniel, Horst and Peter!
```

See also:

Signatures

Continue with *Defining Options*.

1.2.3 Defining Options

So far we have concerned ourselves with whom we greet and not with how, let us change. How an application does things can usually be changed by using options, unfortunately there is not much you can change when saying hello to someone, so we are going to turn our application into something much more general, that is an application that greets people:

```
@application.option('--greeting greeting')
def greeting(context, greeting):
    context['greeting'] = greeting
```

This is quite similar to how main functions are defined. The difference is that the signature in case of an option, includes the name of an option.

Another thing we do is use the *context* object that we have ignored so far. This object is basically a dictionary that is passed around to capture the state of the application. You are supposed to use it to store information gathered in options.

Now we can use this information in the *context* in a main function:

```
@application.main('[name...]')
def main(context, name=None):
    if name is None:
        name = [u'World']
    greeting = context.get('greeting', u'Hello')
    if len(name) == 1:
        print(u'%s, %s!' % (greeting, name[0]))
    elif len(name) == 2:
        print(u'%s %s and %s!' % (greeting, name[0], name[1]))
    else:
        print(u'%s %s and %s!' % (greeting, u', '.join(name[:-1]), name[-1]))
```

Now we can use the *–greeting* option to change the way our application greets people:

```
$ python hello.py --greeting Hi
Hi, World!
```

Continue with *Defining Commands*.

1.2.4 Defining Commands

If you develop larger command line applications, like pip which you probably used to install Argvard, your application often does not perform a specific action like greeting someone but instead allows the user to perform multiple distinct actions.

In the case of pip such an action is installing or uninstalling something. These are fundamentally different actions, not just in what they operate on, how they operate but in what they do. We do not want to choose between these actions based on positional arguments or options, we need a different way to express these: commands.

A `Command` is very much like an argvard object. A command requires a main function that performs something and we can register options on a command.

Let us create a simple calculator as an example:

```
from argvard import Argvard, Command

application = Argvard()

@application.main()
def main(context):
    context.argvard(context.command_path + ['--help'])
```

The calculator, unless called with a command, has nothing useful to do. So within the main function we recursively call the application with the *–help* option, to provide the user with a help message that explains how the application should be used.

Now comes there interesting part, defining the commands:

```
add = Command()
@add.main('a b')
def add_main(context, a, b):
    print(int(a) + int(b))
```

```
sub = Command()
@sub.main('a b')
def sub_main(context, a, b):
    print(int(a) - int(b))
```

As you can see and as mentioned above, command objects are very similar to argvard objects in how they are used. We are not doing this in the tutorial but you could also add options to the *add* or *sub* commands. Now that the commands have been defined you have to register them with the application:

```
application.register_command('add', add)
application.register_command('sub', sub)
```

The string passed to `register_command()` is the name, which is used on the command line to call the command we are registering. In case you were wondering, you can also register commands with other commands.

Finally we call the application, just as we did in our previous "Hello World" application:

```
if __name__ == '__main__':
    application()
```

If you run the application the commands can be invoked as expected:

```
$ python calc.py add 1 1
2
$ python calc.py sub 1 1
0
```

Congratulations! You have now learned everything you need to know about Argvard, to create command line applications.

1.3 Signatures

Signatures are used to define positional arguments of an option, command or the application. A signature consists of zero or more *words* separated by spaces.

A *word* can be a *name*, – and if the signature does not describe the positional arguments of an option – a *repetition* or an *optional*.

A *name* is a python identifier, that an argument will be bound to.

A *repetition* is a name followed by ..., it matches one or more arguments, all of which will be bound to the name.

An *optional* is a name or repetition followed by zero or more words enclosed in brackets.

For a short overview this is the grammar in EBNF:

```
signature = [ word { " " word } ]
word = name | repetition | optional ;
name = (* Any valid Python identifier *) ;
repetition = name "..." ;
optional = "[" (name | repetition) { word } "]" ;
```

1.4 Arguments

1.4.1 Validation with annotations

Argvard can use function annotations to validate and convert arguments:

```
from argvard import Argvard, annotations

application = Argvard()

@application.main('number')
def main(number:float):
    assert isinstance(number, float)
    print('Hooray!')
```

Since Python 2 doesn't have function annotations, you can explicitly use the `argvard.annotations()` function decorator:

```
from argvard import Argvard, annotations

application = Argvard()

@application.main('number')
@annotations(number=float)
def main(number):
    ...
```

Any callable which returns the new value or raises `ValueError` on invalid input can be used instead of `float`.

Most builtin types should just work, but some of them are special-cased to be more liberal in input, and to provide nicer error messages:

- `bool`: Accepts `{"y", "yes", "true"}` for `True` and `{"n", "no", "false"}` for `False`. Case-insensitive.
- `float, int`: Same accepted values as the builtins, but nicer error messages.

If you want `number` to be an optional argument, you would have to write it like this:

```
@application.main('number')
@annotations()
def main(number:float = 1.0):
    ...
```

The `argvard.annotations()` decorator also guesses the type of variables by their default value. In the above example you wouldn't have to specify `number` to be a float:

```
@application.main('number')
@annotations()
def main(number=1.0):
    ...
```

1.4.2 Simple validation

If for some reason you don't want to or can't use annotations, you can still do what the decorator does under the hood and raise `argvard.UsageError` in your functions to show a error message and exit:

```
from argvard import Argvard, UsageError

application = Argvard()

@application.main('number')
def main(number):
    try:
        number = float(number)
    except ValueError:
        raise UsageError('This is not a number.')
```

CHAPTER 2

API Reference

2.1 API

`argvard.__version__`
The version as a string.

`argvard.__version_info__`
The version as a tuple, containing the major, minor, and bugfix version. You should use this, if you need to implement any version checks.

2.1.1 Application Object

class `argvard.`**`Argvard`**(*defaults=None*)
The argvard object is the central object of the command line application.

Instances are callable with the command line arguments, `sys.argv` by default.

The object acts as a registry for options and commands and calls them as necessary.

Parameters **defaults** – A dictionary containing the initial values for the *context*.

classmethod **`from_main`**(*signature=''*)
A decorator that creates an instance and registers the decorated function as main function, see `main()` for more information.

New in version 0.2.

`main`(*signature=''*)
A decorator that is used to register the main function with the given *signature*:

```
@app.main()
def main(context):
    # do something
    pass
```

The main function is called, after any options and if no command has been called.

`option`(*signature*, *overrideable=False*)
A decorator for registering an option with the given *signature*:

```
@app.option('--option')
def option(context):
    # do something
    pass
```

If the name in the signature has already been used to register an option, a `RuntimeError` is raised unless the registered option has been defined with *overrideable* set to *True*.

Parameters

- **signature** – The signature of the option as a string.
- **overrideable** – If *True* the registered option can be overridden.

register_command(*name*, *command*)
Registers the *command* with the given *name*.

If the *name* has already been used to register a command a `RuntimeError` will be raised.

2.1.2 Command Object

class `argvard.`**Command**(*defaults=None*)
A command - like an argvard object - is a registry of options and commands, that represents a distinct action.

Commands can be registered with any number of argvard objects, any number of times (under different names.)

Parameters **defaults** – A dictionary containing the initial values for the *context*, any values already contained in the context once the command is called will not be overridden with a default value.

classmethod from_main(*signature=''*)
A decorator that creates an instance and registers the decorated function as main function, see `main()` for more information.

New in version 0.2.

main(*signature=''*)
A decorator that is used to register the main function with the given *signature*:

```
@app.main()
def main(context):
    # do something
    pass
```

The main function is called, after any options and if no command has been called.

option(*signature*, *overrideable=False*)
A decorator for registering an option with the given *signature*:

```
@app.option('--option')
def option(context):
    # do something
    pass
```

If the name in the signature has already been used to register an option, a `RuntimeError` is raised unless the registered option has been defined with *overrideable* set to *True*.

Parameters

- **signature** – The signature of the option as a string.
- **overrideable** – If *True* the registered option can be overridden.

register_command(*name*, *command*)
Registers the *command* with the given *name*.

If the *name* has already been used to register a command a `RuntimeError` will be raised.

2.1.3 Context Object

class argvard.**Context**(*argvard*, *application_name*)

The context object is a dictionary, passed to options and main functions, which they can use to store information.

It further provides information useful for introspection and debugging through attributes.

argvard

The current application.

command

The current command or *None*.

command_path

A list containing the name of the application and the names of all commands called so far.

caller

The current command or argvard object.

2.1.4 Annotations

argvard.**annotations**(*from_defaults=True*, ***kwargs*)

A function decorator which coerces argument values to the annotated types. This decorator is implicitly applied to command and option functions. Explicitly wrapping your functions with it allows more fine-grained configuration and the usage of annotations in Python 2. Applying this decorator multiple times will raise a RuntimeError.

Parameters

- **from_defaults** – Infer the type of arguments by their default value.
- **kwargs** – Python 2 doesn't have function annotations, so you can also pass types here as keyword arguments.

2.1.5 Exceptions

class argvard.**UsageError**

The user used the program or command in a wrong way.

Raise this exception inside your functions to make argvard display the message, show the help page and exit.

CHAPTER 3

Additional Information

3.1 License

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices

```
        stating that You changed the files; and

    (c) You must retain, in the Source form of any Derivative Works
        that You distribute, all copyright, patent, trademark, and
        attribution notices from the Source form of the Work,
        excluding those notices that do not pertain to any part of
        the Derivative Works; and

    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
```

```
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

3.2 Changelog

3.2.1 Version 0.3.1

In development

- Added Python 3.4 support. It should have worked previously but from now on it's tested.
- Fix an issue that caused docstrings to not be properly dedented when used as descriptions, producing among

other things badly formatted help text.

- Added ability to raise `argvard.UsageError` inside functions to get help output.
- Added *annotations* as a way of validating user input.

3.2.2 Version 0.3.0

- Execution is now delegated to `--help`, if no main function has been defined.

3.2.3 Version 0.2.1

- Fixed typos in the documentation.

3.2.4 Version 0.2.0

- Added `argvard.Argvard.from_main` and `argvard.Command.from_main` to reduce overhead when creating simple applications or commands.

3.2.5 Version 0.1.0

Initial release.

a