Manage Documentation

Release 0.1.13

Bruno Rocha

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Manage

Command Line Manager + Interactive Shell for Python Projects

- Free software: ISC license
- Documentation: https://manage.readthedocs.io.

Features

With **manage** you add a **command line manager** to your Python project and also it comes with an interactive shell with iPython support.

All you have to do is **init** your project directory (creating the manage.yml file)

```
$ pip install manage
$ cd /my_project_root_folder
$ manage init
creating manage.yml....
```

The file **manage.yml** describes how **manage** command should discover your app modules and custom commands and also it defines which objects should be loaded in to the **shell**

Note: Windows users may need to install proper version of PyYAML depending on the version of that thing you call an operating system, installable available in: https://pypi.python.org/pypi/PyYAML **or consider using Linux and don't worry about this as everything works well in Linux except games, photoshop and solitary game :)**

The Shell

By default the command `manage shell` is included, it is a simple Python REPL console with some configurable options.

You can change the banner message to say anything you want, **e.g: "Welcome to my shell!"** and you can also specify some objects to be automatically imported in to the shell context so when you enter in to the shell you already have your project's common objects available.

Also you can specify a custom function to run or a string based code block to run, useful to init and configure the objects.

Consoles

`manage shell` can start different consoles by passing the options

- `manage shell --ipython` - This is the default (if ipython installed)
- `manage shell --ptpython`
- `manage shell --bpython`
- `manage shell --python` - This is the **default Python console** including support for autocomplete. (will be default when no other is installed)

The first thing you can do with **manage** is customizing the objects that will be automatically loaded in to shell, saving you from importing and initializing a lot of stuff every time you need to play with your app via console.

Edit **manage.yml** with:

```
project_name: My Awesome Project
help_text: |
  This is the {project_name} interactive shell!
shell:
  console: bpython
  readline_enabled: false  # MacOS has no readline completion support
  banner:
    enabled: true
    message: 'Welcome to {project_name} shell!'
  auto_import:
    display: true
    objects:
      my_system.config.settings:
      my_system.my_module.MyClass:
      my_system.my_module.OtherClass:
        as: NiceClass
      sys.path:
        as: sp
        init:
          insert:
            args:
              - 0
              - /path/to/be/added/automatically/to/sys/path
  init_script: |
    from my_system.config import settings
    print("Initializing settings...")
    settings.configure()
```

Then the above **manage.yaml** will give you a shell like this:

```
$ manage shell
Initializing settings...
Welcome to My Awesome Project shell!
    Auto imported: ['sp', 'settings', 'MyClass', 'NiceCLass']
>>>  NiceClass. <tab> # autocomplete enabled
```

Watch the demo:

Check more examples in:

https://github.com/rochacbruno/manage/tree/master/examples/

The famous **naval fate** example (used in docopt and click) is in:

https://github.com/rochacbruno/manage/tree/master/examples/naval/

Projects using manage

- Quokka CMS (A Flask based CMS) is using manage
- Red Hat Satellite QE tesitng framework (robottelo) is using manage

Custom Commands

Sometimes you need to add custom commands in to your project e.g: A command to add users to your system:

```
$ manage create_user --name=Bruno --passwd=1234
Creating the user...
```

manage has some different ways for you to define custom commands, you can use **click commands** defined in your project modules, you can also use **function_commands** defined anywhere in your project, and if really needed can define **inline_commands** inside the **manage.yml** file

1. Using a custom click_commands module (single file)

Lets say you have a commands module in your application, you write your custom command there and **manage** will load it

```
# myproject/commands.py
import click
@click.command()
@click.option('--name')
@click.option('--passwd')
def create_user(name, passwd):
    """Create a new user"""
    click.echo('Creating the user...')
    mysystem.User.create(name, password)
```

Now you go to your **manage.yml** or **.manage.yml** and specify your custom command module.

```
click_commands:
  - module: commands
```

Now you run **manage –help**

```
$ manage --help
...
Commands:
  create_user  Create a new user
  debug        Shows the parsed manage file
  init         Initialize a manage shell in current...
  shell        Runs a Python shell with context
```

Using a click_commands package (multiple files)

It is common to have different files to hold your commands so you may prefer having a **commands/** package and some **python** modules inside it to hold commands.

```
# myproject/commands/user.py
import click
@click.command()
@click.option('--name')
@click.option('--passwd')
def create_user(name, passwd):
    """Create a new user"""
    click.echo('Creating the user...')
    mysystem.User.create(name, password)
```

```
# myproject/commands/system.py
import click
@click.command()
def clear_cache():
    """Clear the system cache"""
    click.echo('The cache will be erased...')
    mysystem.cache.clear()
```

So now you want to add all those commands to your **manage** editing your manage file with.

```
click_commands:
  - module: commands
```

Now you run **manage –help** and you have commands from both modules

```
$ manage --help
...
Commands:
  create_user  Create a new user
  clear_cache  Clear the system cache
  debug        Shows the parsed manage file
  init         Initialize a manage shell in current...
  shell        Runs a Python shell with context
```

Custom click_command names

Sometimes the name of commands differ from the name of the function so you can customize it.

```
click_commands:
  - module: commands.system
    config:
      clear_cache:
        name: reset_cache
        help_text: This resets the cache
  - module: commands.user
    config:
      create_user:
        name: new_user
        help_text: This creates new user
```

Having different namespaces

If customizing the name looks too much work for you, and you are only trying to handle naming conflicts you can user namespaced commands.

```
namespaced: true
click_commands:
  - module: commands
```

Now you run **manage –help** and you can see all the commands in the same module will be namespaced by **module-name_**

```
$ manage --help
...
Commands:
  user_create_user    Create a new user
  system_clear_cache  Clear the system cache
  debug         Shows the parsed manage file
  init          Initialize a manage shell in current...
  shell         Runs a Python shell with context
```

And you can even customize namespace for each module separately

Note: If **namespaced** is true all commands will be namespaced, set it to false in order to define separately

```
click_commands:
  - module: commands.system
    namespace: sys
  - module: commands.user
    namespace: user
```

Now you run **manage –help** and you can see all the commands in the same module will be namespaced.

```
$ manage --help
...
Commands:
  user_create_user  Create a new user
  sys_clear_cache  Clear the system cache
  debug         Shows the parsed manage file
  init          Initialize a manage shell in current...
  shell         Runs a Python shell with context
```

2. Defining your inline commands in manage file directly

Sometimes your command is so simple that you do not want (or can't) have a custom module, so you can put all your commands in yaml file directly.

```
inline_commands:
  - name: clear_cache
    help_text: Executes inline code to clear the cache
    context:
      - sys
      - pprint
```

```
    options:
      --days:
        default: 100
    code: |
      pprint.pprint({'clean_days': days, 'path': sys.path})
```

Now running **manage –help**

```
$ manage --help
...
Commands:
  clear_cache  Executes inline code to clear the cache
  debug        Shows the parsed manage file
  init         Initialize a manage shell in current...
  shell        Runs a Python shell with context
```

And you can run using

```
$ manage clear_cache --days 15
```

3. Using general functions as commands

And if you already has some defined function (any callable works).

```
# my_system.functions.py
def create_user(name, password):
    print("Creating user %s" % name)
```

```
function_commands:
  - function: my_system.functions.create_user
    name: new_user
    help_text: Create new user
    options:
      --name:
        required: true
      --password:
        required: true
```

Now running **manage –help**

```
$ manage --help
...
Commands:
  new_user     Create new user
  debug        Shows the parsed manage file
  init         Initialize a manage shell in current...
  shell        Runs a Python shell with context

$ manage new_user --name=Bruno --password=1234
Creating user Bruno
```

Further Explanations

- You can say, **how this is useful?**, There's no need to get a separate package and configure everything in yaml, just use iPython to do it. Besides, IPython configuration has a lot more options and capabilities.

- So I say: Nice! **If you don't like it, dont't use it!**

Credits

- This is inspired by **Django's manage.py command**
- This is based on click
- This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

Similar projects

- Cobra is a *manage* for Go language https://github.com/spf13/cobra

CHAPTER 2

Installation

Stable release

To install Manage, run this command in your terminal:

```
$ pip install manage
```

This is the preferred method to install Manage, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for Manage can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/rochacbruno/manage
```

Or download the tarball:

```
$ curl  -OL https://github.com/rochacbruno/manage/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

This is the SIMPLE example, it is just a folder with a `manage.yml`

```
project_name: My Simple Project
help_text: |
  This is the {project_name} interactive shell
  You can have commands or open the shell
shell:
  readline_enabled: true
  banner:
    enabled: true
    message: 'Hello {project_name} World'
  auto_import:
    display: true
    objects:
      manage.utils.import_string:
      os.path:
        as: path
        init:
          exists:
            kwargs:
              path: /tmp
        init_script: |
          print("path object is:")
          print(type(path))
          print("Hello path from init_script")
      sys.path:
        as: sp
        init:
          insert:
            args:
              - 0
              - /tmp/add_on_object_init
        init_script: |
          def function():
              assert isinstance(sp, list)
```

```
            return type(sp)
        print(function())
  init:
    sys.path.append:
      args:
        - /tmp/added_on_shell_init
  init_script: |
    # add a path to sys.path
    import sys
    sys.path.append('/tmp/added_on_shell_init_script')
    assert '/tmp/added_on_shell_init' in sys.path
    assert '/tmp/add_on_object_init' in sys.path
    assert '/tmp/added_on_shell_init_script' in sys.path
```

and it can be used as:

```
$ pip install manage
```

```
$ cd examples/simple/
$ manage --help
Usage: manage [OPTIONS] COMMAND [ARGS]...

This is the My Simple Project interactive shell You can have commands or
open the shell

Options:
  --help  Show this message and exit.

Commands:
  debug  Shows the parsed manage file
  init   Initialize a manage shell in current...
  shell  Runs a Python shell with context
```

And the shell according to defined attributes in **manage.yml**:

```
$ manage shell
<type 'list'>
path object is:
<type 'module'>
Hello path from init_script
Python 2.7.11 (default, Mar 31 2016, 20:46:51)
IPython 4.2.0 -- An enhanced Interactive Python.
...

Hello My Simple Project World
  Auto imported: ['import_string', 'path', 'function', 'sp']

In [1]:
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/rochacbruno/manage/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

Write Documentation

Manage could always use more documentation, whether as part of the official Manage docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/rochacbruno/manage/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *manage* for local development.

1. Fork the *manage* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/manage.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv manage
$ cd manage/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 manage tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/rochacbruno/manage/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test tests.test_manage
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search