opyrant Documentation

Release 0.1.3

Justin Kiggins

July 20, 2016

Contents

1 opyrant **3**
- 1.1 Features . . . 3
- 1.2 Credits . . . 3

2 Installation **5**
- 2.1 Stable release . . . 5
- 2.2 From sources . . . 5

3 Usage **7**

4 Contributing **9**
- 4.1 Types of Contributions . . . 9
- 4.2 Get Started! . . . 10
- 4.3 Pull Request Guidelines . . . 10
- 4.4 Tips . . . 11

5 Operant logic is easy **13**

6 Writing operant protocols should be easy, but in practice... **15**

7 A better way **17**

8 Documentation **19**

9 Architecture **21**
- 9.1 Behaviors . . . 21
- 9.2 Panels . . . 21
- 9.3 Components . . . 21
- 9.4 Hardware IO Classes . . . 21
- 9.5 Hardware interfaces . . . 22

10 Indices and tables **23**

Contents:

CHAPTER 1

opyrant

hardware abstraction and shareable protocols for operant conditioning

- Free software: BSD license
- Documentation: https://opyrant.readthedocs.io.

1.1 Features

- TODO

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install opyrant, run this command in your terminal:

```
$ pip install opyrant
```

This is the preferred method to install opyrant, 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.

2.2 From sources

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

You can either clone the public repository:

```
$ git clone git://github.com/neuromusic/opyrant
```

Or download the tarball:

```
$ curl  -OL https://github.com/neuromusic/opyrant/tarball/master
```

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

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

CHAPTER 3

Usage

To use opyrant in a project:

```
import opyrant
```

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:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/neuromusic/opyrant/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.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

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

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/neuromusic/opyrant/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 :)

4.2 Get Started!

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

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

```
$ git clone git@github.com:your_name_here/opyrant.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 opyrant
$ cd opyrant/
$ 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 opyrant 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.

4.3 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/neuromusic/opyrant/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_opyrant
```

Pyoperant is a framework to easily construct and share new operant behavior paradigms.

With PyOperant, you can write a single behavior script that works across different species, different computers, different hardware, different rewards, different modalities.

CHAPTER 5

Operant logic is easy

1. Present a stimulus
2. Get the subject's response
3. If the response matches the stimulus, then reward the subject

CHAPTER 6

Writing operant protocols should be easy, but in practice...

Error checking, data storage, and machine-specific hardware interactions often obfuscate the simplicity of the task, limiting its flexibility and power. This limitation becomes increasingly apparent when deploying high-throughput behavioral experiment control systems, transferring subjects from a training panel to an electrophysiology panel, or simply trying to share behavioral protocols.

CHAPTER 7

A better way

PyOperant deals with these challenges by providing a cross-platform object-oriented framework to easily construct, conveniently share, and rapidly iterate on new operant behavior paradigms.

1. Abstract physical component manipulation from low-level hardware manipulation
2. Define behavioral protocols as classes which can be extended through object inheritance

Further, experimenters are able to integrate their behavioral protocols with other Python packages for online data analysis or experimental control. We currently use pyoperant in the Gentner Lab to control 36 operant panels.

CHAPTER 8

Documentation

PyOperant abstracts behavioral protocol logic from hardware interactions through a machine-specific configuration file. In the local.py configuration file, the experimenter defines the operant panels available for use. A Panel consists of a collection of Component objects and a set of standard methods to manipulate the Component. These Component objects are mirrors of their physical counterparts, such as a food hopper, response port, speaker, or house light.

Behavioral protocols can be modifed and extended through object inheritance. The modular architecture of PyOperant also allows experimenters to integrate their behavioral protocols with other Python packages for online data analysis or experimental control.

PyOperant's hardware support currently includes PortAudio & Comedi. Future support will include NiDAQmx and Cambridge Electronic Designs.

CHAPTER 9

Architecture

9.1 Behaviors

Behaviors are Python classes which run the operant experiment. They associate the subject with the hardware panel the subject is interacting with and save experimental data appropriately. They are instantiated with various experimental parameters, such as stimulus identities and associations, block designs, and reinforcement schedules.

There are a couple of built-in behaviors: TwoAltChoice, which runs two alternative choice tasks and Lights, which simply turns the house light on and off according to a schedule. These can be inherited to change specific methods without changing the rest of the behavioral protocol.

9.2 Panels

Panels are the highest level of hardware abstraction. They maintain panel components as attributes and have standard methods for resetting and testing the panel. Many Behaviors rely on specific panel components and methods to be present.

Panels are defined by the experimenter locally.

9.3 Components

Components are common hardware components, such as a Hopper, a ResponsePort, a HouseLight, or an RGBLight. Many components rely on multiple hardware IO channels. For example, a Hopper requires both a solenoid (to activate the Hopper) and an IR beam detector (to check if the Hopper is raised). Calling the 'feed' method on a Hopper checks to make sure that the hopper is down, raises the hopper, checks to make sure the hopper raised, waits the appropriate length of time, then lowers the hopper, finally checking one more time to make sure the hopper dropped. If there is an incongruity between the status of the solenoid and the IR beam, the Hopper component raises the appropriate error, which the Behavior script can deal with appropriately.

9.4 Hardware IO Classes

Hawdware IO classes standardize inputs and outputs that are available for Components and Panels to use.

9.5 Hardware interfaces

Hardware interfaces are wrappers around hardware drivers and APIs that allow hardware IO classes to work.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search