Labware Index

Release 1.0.0

Kenny Workman

Dec 30, 2019

CONTENTS:

CHAPTER

ONE

QUICKSTART AND UNIT TESTING

1.1 Download

Setting up the Labware Index tool should be a relatively quick process.

- Grab the source code from its home on git

```
git clone https://github.com/kennyworkman/labware-index
```

- Navigate to the project direcotry and build *pyindex* as a local package using the *setup.py* file in the repo. (Activate a virtual environment if desired):

```
cd labware-index
python3 setup.py install
```

- Download extraneous dependencies for testing and using the GUI:

```
pip3 install -r requirements.txt
```

That's it. You should be all set!

1.2 Testing

Unit testing is implemented with *pytest*. Every method in the *pyindex* package was implemented in isolation using a rigorous test-driven approach, guaranteeing essential functionality and safe-guarding against potential edge cases.

To run the suite of unit tests:

- Navigate to the *tests* directory and run pytest:

```
cd tests
pytest
```

Verbose feedback from the testing suite should be provided in terminal. If tests fail, make sure the Download steps were followed correctly. Consider rebuilding the package from source. *pytest* uses the locally built package sitting in your virtual environment to run its testing suite.

CHAPTER

TWO

API REFERENCE

What follows is a thorough specification of the entire pyindex API; an in-depth breakdown of the application interface, highlighting certain design choices in class architecture.

2.1 The Registry

This is the primary user-facing interface for pyindex. It handles the indexing, storing, and retrieval of Labware. The bulk of tool's functionality can be extrapolated from the handful of methods within this class.

In the abstract, there should really only be a single *Registry* in existence at any given time. The *Registry* constructor thus will "load" a *Registry* from file when a user attempts to instantiate one in the presence of a directory with persisted data. In such a scenario, a new *Registry* will be created to reflect the state of this serialized data, and *Existing registry loaded from disk* will be printed to std out to give off the illusion of a Singleton pattern.

The design choice for persistence was largely inspired by git. Labware objects are serialized, using the *pickle* package that comes in the Python standard library, into files uniquely hashed by object content. The crytographic hashing function used is Secure Hash 1, or **SHA-1**, and produces a 160-bit integer hash from a *Labware* object byte stream. Hashing collisions under this algorithm are impossible in practice, making for efficient and constant time access to our data. (This is similar to the storage of commit and blob objects in the *.git* folder for git version control.)

Users are able to index their *Labware* types by "nicknames", or whatever custom naming scheme they desire. A serialized mapping of these nicknames to object hash IDs is maintained to facilitate constant time object access. The choice to serialize objects by SHA-1 hashcodes allows persisted objects to be strictly decoupled from potentially frivolous and arbitrary user naming schemes, avoiding the mapping of identical objects to different hashes and vice-versa, while still allowing users to customize their Labware Index.

More information about persistence design choices are in the *Registry* section of the API.

2.1.1 registry.py

Defines the Registry class.

class pyindex.registry.**Registry**

The Registry for Labware types.

Persistence is maintained by a file system organized as follows:

- A directory located in the **current working directory** named

  ./.labware.

- Within this directory, serialized Labware objects are stored in files

named after their respective Secure Hash Algorithm (SHA-1) ids.

- A serialized mapping of unique hash ids to user-defined names is

  kept in *./.labware/index*

NOTE: If you built pyindex as a package, the current working directory will be somewhere in your site-packages directory (probably associated with some virtual environment), as will your persisted data. Rebuilding pyindex will *delete all such files*.

__eq__(*other*)
Identical attributes between objects is sufficient for equality.

__init__()
Creates a fresh Registry in the current working directory.

The *existence* of a Registry is defined simply by a *.labware* directory in the current working directory; attempting to instantiate a Registry with this directory present will "load" this data into the new object.

Deleting this *.labware* folder will permanently wipe data from the indexing tool. A new Registry can be created at this point.

__repr__()
Representation of the Registry

__weakref__
list of weak references to the object (if defined)

add(*labware*, *name=None*)
Adds a Labware object to the Registry.

This method is intended to be used internally. While one could construct a Labware object, the use of a JSON object or file with the *add_json()* or *add_file()* methods is probably more convenient and easier to use. A JSON template is provided in the *Labware* class documentation.

Parameters

- **name** (*str*) – User-defined name to identify the labware.
- **labware** (Labware) – The Labware object being indexed.

add_file(*name*, *file*)
Adds a Labware object to the Registry using a .json file.

Parameters

- **name** (*str*) – User-defined name to identify the labware.
- **file** – A path to a file containing valid JSON data (validity

specified in *add_json* function) :type file: str

add_json(*name*, *json_data*)
Adds a Labware object to the Registry using raw JSON data.

Parameters

- **name** (*str*) – User-defined name to identify the labware.
- **json_data** (*JSON*) – Valid JSON data with correct number and type of fields. **Note**: thorough documentation of what constitutes *valid* can be found in the *Labware* class.

get(*name*)
Retrieves a Labware object from the Registry.

Parameters **name** (*str*) – The user-defined name associated with the desired Labware type.

Returns The desired Labware type.

Return type Labware.

list()
List the Labware types currently indexed by user-defined names.

Returns A list of user-defined names.

Return type list

remove(*name*)
Removes a Labware object from the Registry by name.

Parameters **name** (*str*) – The user-defined name associated with the Labware to remove.

wipe()
Removes all data stored in this Registry.

This is a dangerous and irreversible operation.

2.2 Layers Upon Layers

When designing the class types to represent a piece of labware, a concerted effort was made to encapsulate pieces of its functionality into modular and relevant layers of abstraction. The *Labware* type was broken down into further subtypes when its components, the *Plate* and the *Well*, could have utility that is distinct and encapsulated from the *Labware* type as whole. The *Well*, for instance, is the site of a reaction determined solely by its particular attributes, ie. volume, depth, and any reagents that it contains, where as its encompassing "Labware" is more or less a vessel.

This layered structure makes for a rather intuitive interface for attribute access. If one wishes to know how many wells their labware has, they would make the following call:

```
labware.plate.num_wells
```

Likewise, the volume of their well is:

```
labware.well.volume
```

A detailed breakdown of the nested class types continues below.

2.3 The Labware Object

2.3.1 labware.py

Defines the Labware class.

class pyindex.labware.**Labware**(*json_data*)
A class representing a generic SBS-footprint labware type.

__eq__(*other*)
Unique property of SHA-1 will guarantee equality of attributes.

__init__(*json_data*)
Creates a new labware type from valid JSON data.

Valid json data is defined as any such JSON-formatted data type that possess the necessary fields to thoroughly define a Labware object. A template structure is provided below; be sure to include all fields given in the template to avoid an instantiation error:

```
{
    "name": str,
    "plate": {
        "sterile": boolean,
        "skirted": boolean,
        "enzyme_free": boolean,
        "length": float,
        "width": float,
        "height": float,
        "well_spacing": float,
        "well_num": int,
        "composition": boolean (optional),
    },
    "well": {
        "volume": float,
        "depth": float,
        "top_diameter": float,
        "bottom_diameter": float,
    }
}
```

A more complete specifications describing these fields can be found in the parameter descriptions of the *Plate* and *Well* object documentation.

Parameters **json_data** (*JSON*) – Valid JSON data as described above.

__repr__()
Succinct Labware representation.

__weakref__
list of weak references to the object (if defined)

hash()
Generates an SHA-1 hashcode for this object.

Unique generation will avoid potential hashing collissions that emerge from indexing based on user-defined naming schemes.

Returns SHA-1 hashcode.

Return type str

save(*registry*, *name=None*)
Stores a serialized version of self in the Registry.

The process of "saving" is twofold:

- A serialized object is stored in the object directory with a filename

  corresponding to its SHA-1 hash code.

- The index file is updated with a mapping of this object's informal

  name to its hashcode.

  Parameters

- **registry** (Registry) – Target Registry to save this piece of Labware to.
- **name** (*str*) – Optional user-defined name for this piece of Labware in the context of a Registry. This name can be used to access the Labware, and defaults to the name of the object.k

2.4 The Plate Object

2.4.1 plate.py

Defines the Plate class.

class pyindex.plate.**Plate**(*sterile*, *skirted*, *enzyme_free*, *length*, *width*, *height*, *well_spacing*, *well_num*, *well*, *composition='unknown'*)

The representation of the Plate infrastructure for arbitrary Labware.

__eq__(*other*)

Identical attributes between objects is sufficient for equality.

__init__(*sterile*, *skirted*, *enzyme_free*, *length*, *width*, *height*, *well_spacing*, *well_num*, *well*, *composition='unknown'*)

Defines new Plate infrastructure.

Parameters

- **sterile** (*bool*) – True if the plate is sterile.
- **skirted** (*bool*) – True if the plate is skirted (has a lip around the base).
- **enzyme_free** (*bool*) – True if the plate is tested to be free of DNase/RNase.
- **composition** (*str*) – Description of the composition material. The only optional parameter. Defaults to "unknown" if nothing is provided.
- **length** (*float*) – Length of the plate *at the base* in **mm**.
- **width** (*float*) – Width of the plate *at the base* in **mm**.
- **height** (*float*) – Overall plate height in **mm**.
- **well_spacing** (*float*) – Distance between *the centers* of any two wells in **mm**.
- **well_num** (*int*) – Number of wells.
- **well** (Well) – Type of well used in the plate.

__repr__()

Succinct Plate representation.

__weakref__

list of weak references to the object (if defined)

2.5 The Well Object

2.5.1 well.py

Defines the Well class.

class pyindex.well.**Well**(*volume*, *depth*, *top_diameter*, *bottom_diameter*)
The representation of a single well within some plate.

__eq__(*other*)
Identical attributes between objects is sufficient for equality.

__init__(*volume*, *depth*, *top_diameter*, *bottom_diameter*)
Defines a new well.

Parameters

- **volume** (*float*) – The *maximum* working volume for a single well in **uL**
- **depth** (*float.*) – The depth of a well in **mm**.
- **top_diameter** (*float.*) – The diameter at the opening in **mm**.
- **bottom_diameter** (*float.*) – The diameter at the bottom of the conical section in **mm**.

__repr__()
Succinct Well representation.

__weakref__
list of weak references to the object (if defined)

2.6 Some Errors

Two internal Errors were subclassed to provide verbose and specific handling of extraneous circumstances

class pyindex.error.**BadJSONError**
Is indicative of incorrectly structured or incomplete JSON data, for the purpose of insantiating data types.

class pyindex.error.**ExistingRegistryError**
Will be thrown if the user is trying to instantiate a Registry object in a directory where one already exists.

CHAPTER

THREE

USING THE API

Here a suggested workflow using the API with a python interpreter is presented.

First instantiate a fresh *Registry*. Note the format of the object representation.

```
>>> from pyindex.registry import Registry
>>> registry = Registry()
>>> registry
Labware Registry
________________
```

Now we need to add a *Labware* object to our *Registry*. We have three options to do so:

- directly passing a *Labware* object
- providing raw JSON data
- loading JSON data from a file.

Working with JSON files is the cleanest and safest option, especially when using a template or using sample files included with the repository (located under `tests/labware_json/` or `gui/sample_json/`).

```
>>> registry.add_file("RAD", "tests/labware_json/biorad_HSP9601B.json")
```

Now our tool has something in it. Notice the updated object repr.

```
>>> registry
Labware Registry
________________
* RAD --> Bio-Rad-HSP9601B with length 127.76 mm, width 85.48 mm, height 16.06 mm,␣
↪and 96 wells.
```

"RAD" is an example of a user-defined name. You can name your *Labware* however you wish. The serialization of the actual object occurs independently of this field.

Lets add another piece of labware.

```
>>> registry.add_file("CORN", "tests/labware_json/corning_3960.json")
>>> registry
Labware Registry
________________
* RAD --> Bio-Rad-HSP9601B with length 127.76 mm, width 85.48 mm, height 16.06 mm,␣
↪and 96 wells.
* CORN --> Corning 3960 with length 127.8 mm, width 85.9 mm, height 43.8 mm, and 96␣
↪wells.
```

Now that we have some labware filed away. How do we access it?

We can get a quick list of our named labware with:

```
>>> registry.list()
['RAD', 'CORN']
```

Using our names, we can then query an object and pick it apart for a desired attribute(s).

```
>>> rad = registry.get("Rad")
>>> rad
Bio-Rad-HSP9601B with length 127.76 mm, width 85.48 mm, height 16.06 mm, and 96 wells.
>>> rad.plate
Plate with length 127.76 mm, width 85.48 mm, height 16.06 mm, and 96 wells.
>>> rad.well
Well with volume 200 uL, depth 14.81 mm, top diameter 5.46 mm, and bottom diameter 2.
↪64 mm.
>>> rad.plate.well_num
96
```

A detailed look at all of the *Labware* attributes is provided in the API section of the documentation.

Let's say I don't like "CORN" anymore. We can remove it simply as so:

```
>>> registry.remove("CORN")
>>> registry
Labware Registry
________________
* RAD --> Bio-Rad-HSP9601B with length 127.76 mm, width 85.48 mm, height 16.06 mm,␣
↪and 96 wells.
```

To start completely fresh, we can wipe our repo.

```
>>> registry.wipe()
>>> registry
Labware Registry
________________
```

CHAPTER

FOUR

USING THE GUI

The graphical interface is a minimal visual tool that was implemented using the *PySimpleGui* library, a wrapper itself around the more common *tkinter* and *qt* python GUI libraries.

Navigate yourself to the `/gui` directory in the project directory. Make sure the Download steps were followed thoroughly. Opening the GUI is as simple as:

```
python3 gui.py
```

The main "control panel" should pop up.

Here's a brief rundown of the button functionality:

- *Info* - Select/Click on a Labware for a popup of summary characteristics.
- *Remove* - Select/Click on a Labware to permanently delete from the Registry.
- *Add* - Manually fill in characteristics for a new Labware type using a GUI popup. These fields are *not* validated because the dynamic typing of python. Use with care.
- *Add From File* - Load a new Labware type from a valid .json file using a file browser. A directory `sample_json/` is provided within the `gui/` directory with some starter Labware types for this exact purpose.
- *Wipe* - Permanently reset the entire Registry.

To load our sample Labware, we must click on the *Add From File* button, and navigate to the `sample_json/lp_0200.json` (for example), name our Labware and click *Save*.

The rest of the commands should be fairly easy to navigate and follow from the API calls that they encapsulate. There is a *Help* button that summarizes such information if you need it.

CHAPTER

FIVE

EXTENSION

Python was chosen for its speed of development and familiarity. Of course, there are limits to its lack of type enforcement. Here we trust the user to read the documentation and use the methods and objects with care and strict adherence to documented parameter types.

When designing types, there was an interesting problem to define. What is the minimum amount of data to define a piece of labware? What even is labware?

The conclusion was a *Labware* type needs:

- Sufficient information to build a *Well*.
- Dimensional information about its plate and boolean attributes pertaining to sterility

An interesting result was that composition attributes of the *Labware* were often ambiguous as provided by retailers, and impractical to consistently include in *Labware* type construction in any closed form. This obviously limits potential simulated pipeline runs, when reactivity between reagents and labware material is a concern, but is unavoidable due to (maybe proprietary?) opaque access to information.

Because the *Well* sits at the lowest level of abstraction within the Labware object, acting as a decoupled reaction "context" defined by its dimensions, volume, and composition, it would be easy to add attributes describing what exactly is going on in each of the wells. Though it was not done here to minimize memory expense, it would be easy to represent a *Plate* as a multi-dimensional array, where each item in said array references a unique *Well* object that holds reagents and conditions for a unique reaction.

Likewise, might make sense to define different ranges of working volumes for distinct reagents/reactions within the *Well* type, though this would be application specific.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

E

G

H

L

P

R

S

W