Opentrons API Documentation

Release 3.8.1

Opentrons Labworks

Apr 09, 2019

Contents

The Opentrons API is a simple Python framework designed to make writing automated biology lab protocols easy.

We've designed it in a way we hope is accessible to anyone with basic Python and wetlab skills. As a bench scientist, you should be able to code your automated protocols in a way that reads like a lab notebook.

---

1 How it Looks

The design goal of the Opentrons API is to make code readable and easy to understand. For example, below is a short set of instruction to transfer from well `'A1'` to well `'B1'` that even a computer could understand:

```
Use the Opentrons API's labware and instruments

This protocol is by me; it's called Opentrons Protocol Tutorial and is used for␣
↪demonstrating the Opentrons API

Add a 96 well plate, and place it in slot '2' of the robot deck
Add a 200uL tip rack, and place it in slot '1' of the robot deck

Add a single-channel 300uL pipette to the left mount, and tell it to use that tip rack

Transfer 100uL from the plate's 'A1' well to it's 'B2' well
```

If we were to rewrite this with the Opentrons API, it would look like the following:

```
# imports
from opentrons import labware, instruments

# metadata
metadata = {
    'protocolName': 'My Protocol',
    'author': 'Name <email@address.com>',
    'description': 'Simple protocol to get started using OT2',
}

# labware
plate = labware.load('96-flat', '2')
tiprack = labware.load('opentrons-tiprack-300ul', '1')

# pipettes
pipette = instruments.P300_Single(mount='left', tip_racks=[tiprack])

# commands
pipette.transfer(100, plate.wells('A1'), plate.wells('B2'))
```

---

2 How it's Organized

When writing protocols using the Opentrons API, there are generally five sections:

1. *Imports* (page 3)
2. *Metadata* (page 3)
3. *Labware* (page 3)

4. *Pipettes* (page 3)
5. *Commands* (page 3)

2.1 Imports

When writing in Python, you must always include the Opentrons API within your file. We most commonly use the `labware` and `instruments` sections of the API.

From the example above, the "imports" section looked like:

```
from opentrons import labware, instruments
```

2.2 Metadata

Metadata is a dictionary of data that is read by the server and returned to client applications (such as the Opentrons App). It is not needed to run a protocol (and is entirely optional), but if present can help the client application display additional data about the protocol currently being executed.

The fields above ("protocolName", "author", and "description") are the recommended fields, but the metadata dictionary can contain fewer or additional fields as desired (though non-standard fields may not be rendered by the client, depending on how it is designed).

You may see a metadata field called "source" in protocols you download directly from Opentrons. The "source" field is used for anonymously tracking protocol usage if you opt-in to analytics in the Opentrons App. For example, protocols from the Opentrons Protocol Library may have "source" set to "Opentrons Protocol Library". You shouldn't define "source" in your own protocols.

2.3 Labware

While the imports section is usually the same across protocols, the labware section is different depending on the tip racks, well plates, troughs, or tubes you're using on the robot.

Each labware is given a type (ex: `'96-flat'`), and the slot on the robot it will be placed (ex: `'2'`).

From the example above, the "labware" section looked like:

```
plate = labware.load('96-flat', '2')
tiprack = labware.load('tiprack-200ul', '1')
```

2.4 Pipettes

Next, pipettes are created and attached to a specific mount on the OT-2 (`'left'` or `'right'`).

There are other parameters for pipettes, but the most important are the tip rack(s) it will use during the protocol.

From the example above, the "pipettes" section looked like:

```
pipette = instruments.P300_Single(mount='left', tip_racks=[tiprack])
```

2.5 Commands

And finally, the most fun section, the actual protocol commands! The most common commands are `transfer()`, `aspirate()`, `dispense()`, `pick_up_tip()`, `drop_tip()`, and much more.

This section can tend to get long, relative to the complexity of your protocol. However, with a better understanding of Python you can learn to compress and simplify even the most complex-seeming protocols.

From the example above, the "commands" section looked like:

```
pipette.transfer(100, plate.wells('A1'), plate.wells('B1'))
```

Design with Python

Writing protocols in Python requires some up-front design before seeing your liquid handling automation in action. At a high-level, writing protocols with the Opentrons API looks like:

1. Write a Python protocol
2. Test the code for errors
3. Repeat steps 1 & 2
4. Calibrate labware on robot
5. Run your protocol

These sets of documents aim to help you get the most out of steps 1 & 2, the "design" stage.

Python for Beginners

If Python is new to you, we suggest going through a few simple tutorials to acquire a base understanding to build upon. The following tutorials are a great starting point for working with the Opentrons API (from learnpython.org[1]):

1. Hello World[2]
2. Variables and Types[3]
3. Lists[4]
4. Basic Operators[5]
5. Conditions[6]
6. Loops[7]
7. Functions[8]
8. Dictionaries[9]

[1] http://www.learnpython.org/
[2] http://www.learnpython.org/en/Hello%2C_World%21
[3] http://www.learnpython.org/en/Variables_and_Types
[4] http://www.learnpython.org/en/Lists
[5] http://www.learnpython.org/en/Basic_Operators
[6] http://www.learnpython.org/en/Conditions
[7] http://www.learnpython.org/en/Loops
[8] http://www.learnpython.org/en/Functions
[9] http://www.learnpython.org/en/Dictionaries

After going through the above tutorials, you should have enough of an understanding of Python to work with the Opentrons API and start designing your experiments! More detailed information on python can always be found at the python docs[10]

Working with Python

Using a popular and free code editor, like Sublime Text 3[11], is a common method for writing Python protocols. Download onto your computer, and you can now write and save Python scripts.

Note: Make sure that when saving a protocol file, it ends with the `.py` file extension. This will ensure the App and other programs are able to properly read it.

For example, `my_protocol.py`

Simulating Python Protocols

In general, the best way to simulate a protocol is to simply upload it to an OT 2 through the Opentrons app. When you upload a protocol via the Opentrons app, the robot simulates the protocol and the app displays any errors. However, if you want to simulate protocols without being connected to a robot, you can download the Opentrons python package.

Installing

To install the Opentrons package, you must install it from Python's package manager, *pip*. The exact method of installation is slightly different depending on whether you use Jupyter on your computer (note: you do not need to do this if you want to use the *Robot's Jupyter Notebook* (page 6), ONLY for your locally-installed notebook) or not.

Non-Jupyter Installation

First, install Python 3.6 (Windows x64[12], Windows x86[13], OS X[14]) on your local computer.

Once the installer is done, make sure that Python is properly installed by opening a terminal and doing `python --version`. If this is not 3.6.4, you have another version of Python installed; this happens frequently on OS X and sometimes on windows. We recommend using a tool like pyenv[15] to manage multiple Python versions. This is particularly useful on OS X, whch has a built in install of Python 2.7 that should not be removed.

Once python is installed, install the opentrons package[16] using `pip`:

```
pip install opentrons
```

You should see some output that ends with `Successfully installed opentrons-3.6.5` (the version number may be different).

[10] https://docs.python.org/3/index.html
[11] https://www.sublimetext.com/3
[12] https://www.python.org/ftp/python/3.6.4/python-3.6.4-amd64.exe
[13] https://www.python.org/ftp/python/3.6.4/python-3.6.4.exe
[14] https://www.python.org/ftp/python/3.6.4/python-3.6.4-macosx10.6.pkg
[15] https://github.com/pyenv/pyenv
[16] https://pypi.org/project/opentrons/

Jupyter Installation

You must make sure that you install the *opentrons* package for whichever kernel and virtual environment the notebook is using. A generally good way to do this is

```
import sys
!{sys.executable} -m pip install opentrons
```

Simulating

Once the Opentrons Python package is installed, you can simulate protocols in your terminal using the `opentrons_simulate` command:

```
opentrons_simulate.exe my_protocol.py
```

or, on OS X or linux,

```
opentrons_simulate my_protocol.py
```

The simulator will print out a log of the actions the protocol will cause, similar to the Opentrons app; it will also print out any log messages caused by a given command next to that list of actions. If there is a problem with the protocol, the simulation will stop and the error will be printed.

The simulation script can also be invoked through python with `python -m opentrons.simulate /path/to/protocol`.

This also provides an entrypoint to use the Opentrons simulation package from other Python contexts such as an interactive prompt or Jupyter. To simulate a protocol in python, open a file containing a protocol and pass it to `opentrons.simulate.simulate`:

```
import opentrons.simulate
protocol_file = open('/path/to/protocol.py')
runlog = opentrons.simulate.simulate(protocol_file)
print(format_runlog(runlog))
```

The *`opentrons.simulate.simulate()`* (page 74) method does the work of simulating the protocol and returns the run log, which is a list of structured dictionaries. *`opentrons.simulate.format_runlog()`* (page 73) turns that list of dictionaries into a human readable string, which is then printed out. For more information on the protocol simulator, see *Simulation* (page 73).

Configuration and Local Storage

The module uses a folder in your user directory as a place to store and read configuration and changes to its internal data. For instance, if your protocol creates a custom labware, the custom labware will live in the local storage location. This location is `~/.opentrons` on Linux or OSX and `C:\Users\%USERNAME%\.opentrons` on Windows.

Robot's Jupyter Notebook

For a more interactive environment to write and debug using some of our API tools, we recommend using the Jupyter notebook which is installed on the robot. Using this notebook, you can develop a protocol by running its commands line-by-line, ensuring they do exactly what you want, before saving the protocol for later execution.

You can access the robot's Jupyter notebook by following these steps:

1. Open your Opentrons App and look for the IP address of your robot on the robot information page.
2. Type in `(Your Robot's IP Address):48888` into any browser on your computer.

Here, you can select a notebook and develop protocols that will be saved on the robot itself. Note that these protocols will only be on the robot unless specifically downloaded to your computer using the `File / Download As` buttons in the notebook.

Note: When running protocol code in a Jupyter notebook, before executing protocol steps you must call `robot.connect()`:

```
from opentrons import robot
robot.connect()
```

This tells the notebook to connect to the robot's hardware so the commands you enter actually cause the robot to move.

However, this happens automatically when you upload a protocol through the Opentrons app, and connecting twice will cause errors. To avoid this, **remove the call to robot.connect()** before uploading the protocol through the Opentrons app.

Labware

We spend a fair amount of time organizing and counting wells when writing Python protocols. This section describes the different ways we can access wells and groups of wells.

Labware Library

The Opentrons API comes with many common labware built in. These can be loaded into you Python protocol using the `labware.load()` method, and the specific name of the labware you need.

Under the *Opentrons Labware* (page 9) are a list of some of the most commonly used labware in the API, as well as images for how they look.

If you are interested in using your own labware that is not included in the API, please take a look at how to create custom labware definitions using `labware.create()`, or contact Opentrons Support.

Note: All names are case-sensitive, copying and pasting from this list into the protocol editor will ensure no errors are made.

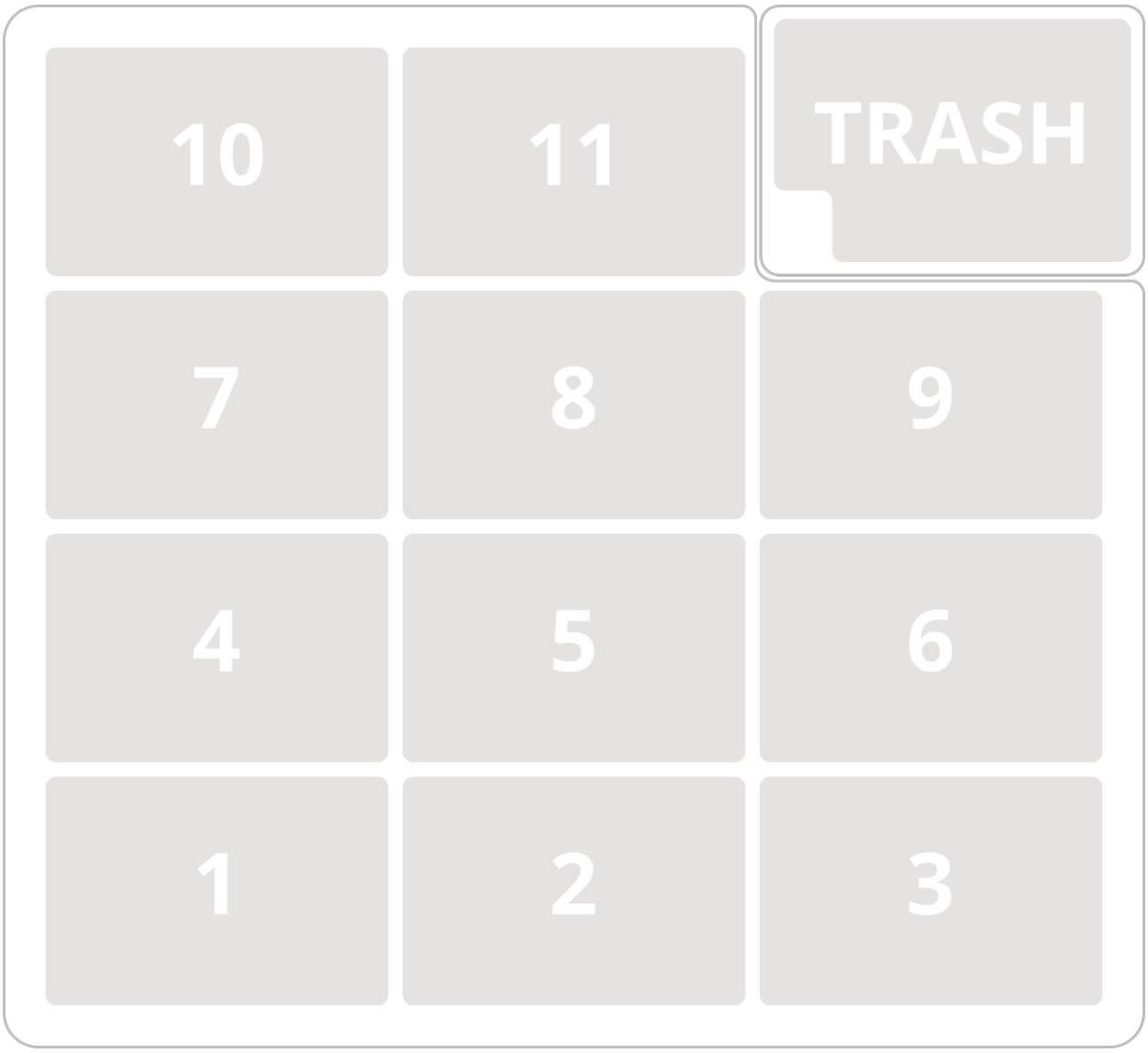
Note: We are in the process of revising the labware definitions used on the OT2. Documentation for previously existing definitions is left over from OT1, and is incomplete. Check out this webpage[17] to see a visualization of all the API's legacy built-in labware definitions. For JSON protocols see the visualizations and descriptions under *Opentrons Labware* (page 9).

---

[17] https://andysigler.github.io/ot-api-containerviz/

Placing labware on the robot deck

The robot deck is made up of slots labeled 1, 2, 3, 4, and so on.



To tell the robot what labware will be on the deck for your protocol, use *labware.load* after importing labware as follows:

```
from opentrons import labware
```

```
samples_rack = labware.load('tube-rack-2ml', slot='1')
```

Opentrons Labware

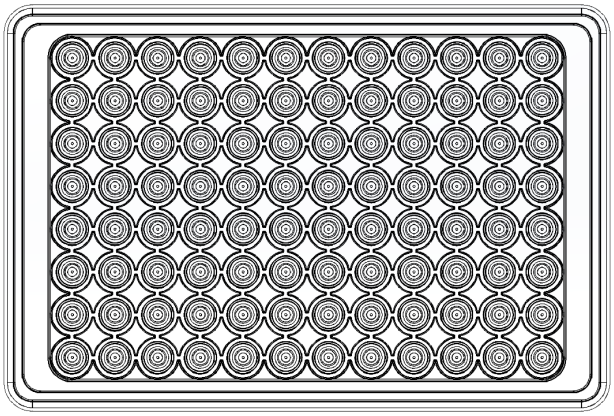
Tipracks

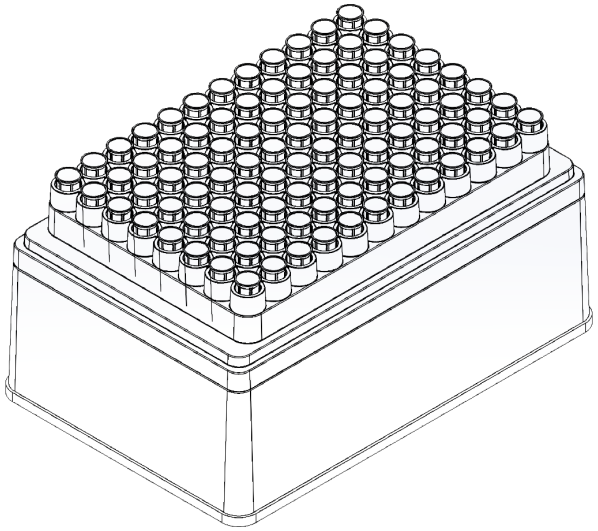
opentrons-tiprack-300ul

Tiprack for both a 50ul and 300ul pipette (single or 8-channel)

```
labware.load('opentrons-tiprack-300ul', slot)
```

Accessing Tips: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`





Aluminum Blocks

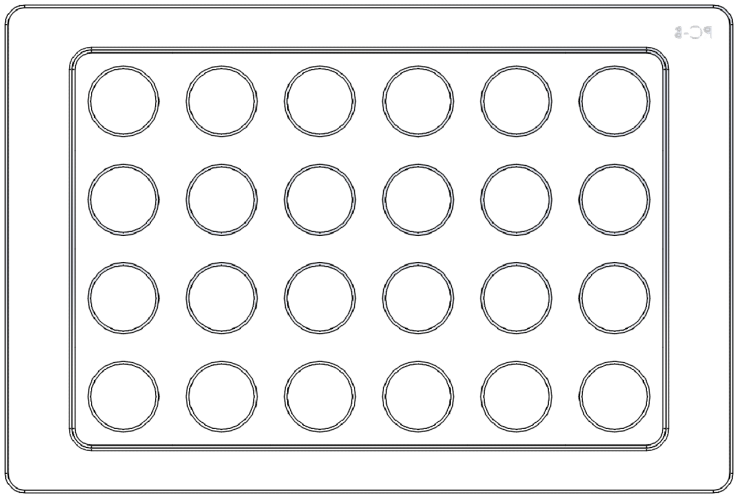
Aluminum blocks are typically paired with a temperature module. Can also be used as a standalone container.

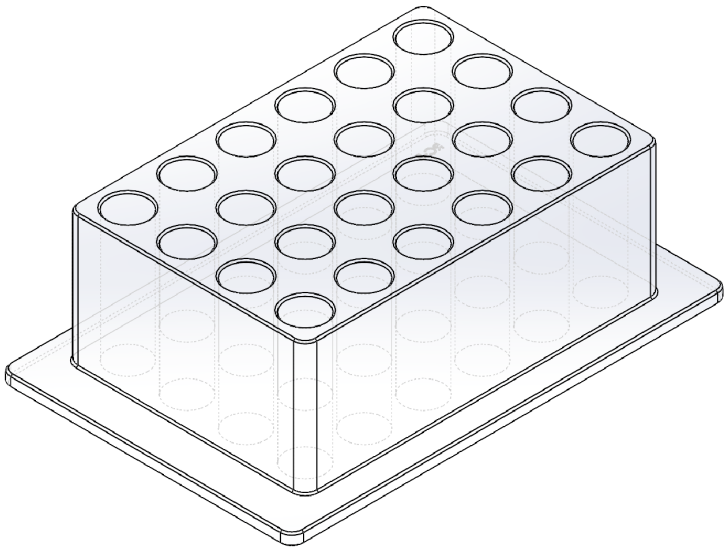
opentrons-aluminum-block-2ml-eppendorf

A 2ml tube holder, specifically based off of eppendorf snapcap tubes.

```
labware.load('opentrons-aluminum-block-2ml-eppendorf', slot)
```

Accessing Wells: *single channel* `['A1']-['D6']`



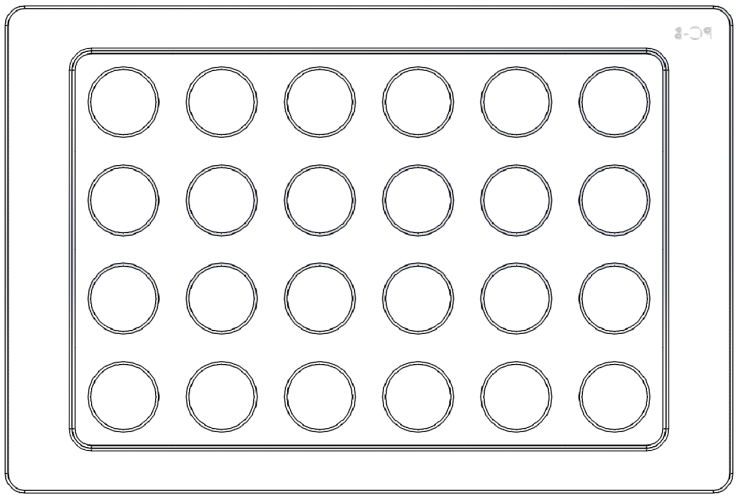


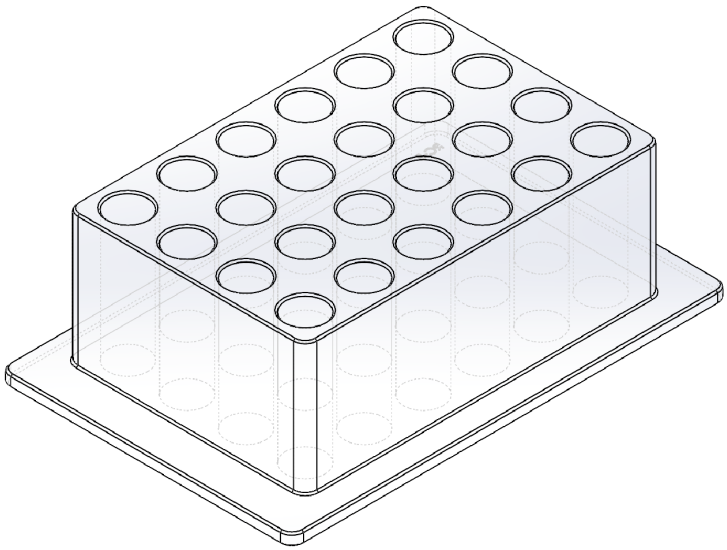
opentrons-aluminum-block-2ml-screwcap

A 2ml tube holder, specifically based off of screwcap tubes

```
labware.load('opentrons-aluminum-block-2ml-screwcap', slot)
```

Accessing Wells: *single channel* `['A1']-['D6']`





opentrons-aluminum-block-96-PCR-plate

A flat plate which acts as an adaptor for a well plate. This particular definition is modeled after the 96-well biorad hardshell plate.

```
labware.load('opentrons-aluminum-block-96-PCR-plate', slot)
```

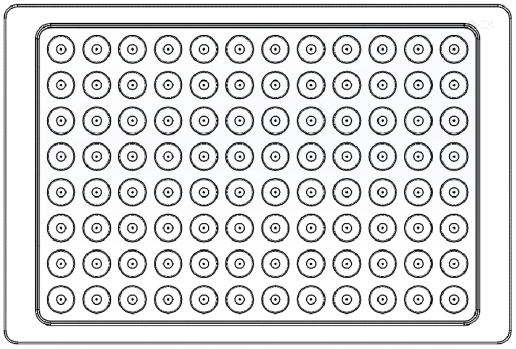
Accessing Wells: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

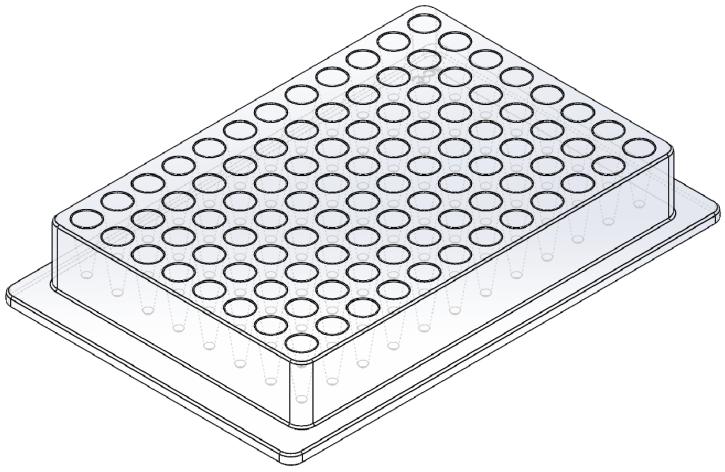
opentrons-aluminum-block-PCR-strips-200ul

A 96 well adaptor meant to hold 96 PCR strips with 200ul max volume.

```
labware.load('opentrons-aluminum-block-PCR-strips-200ul', slot)
```

Accessing Wells: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

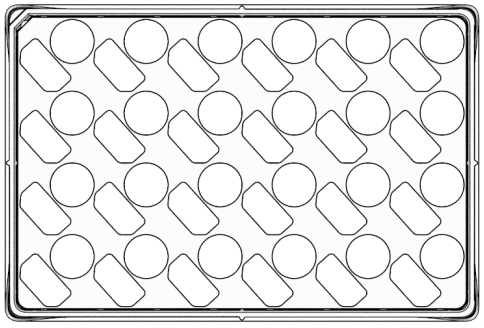


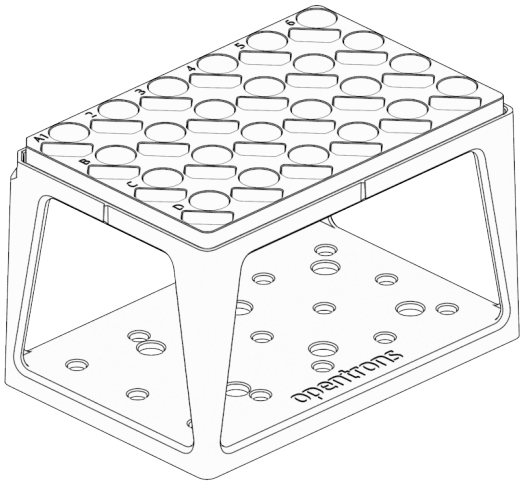


Modular Tuberack

opentrons-tuberack-2ml-eppendorf

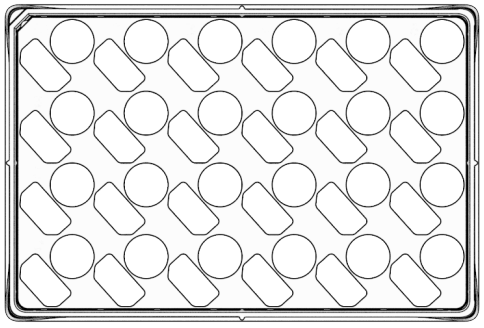
This tuberack insert definition is for snapcap tubes ranging from 2-5ml.

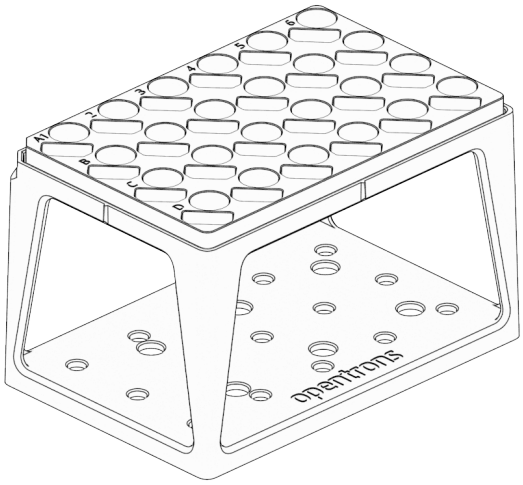




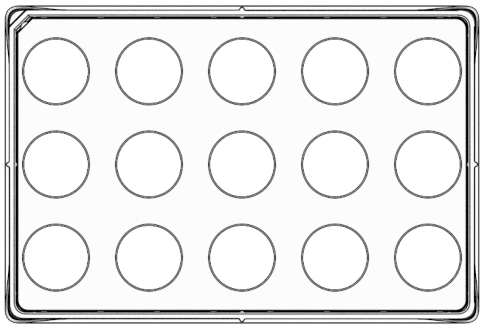
opentrons-tuberack-2ml-screwcap

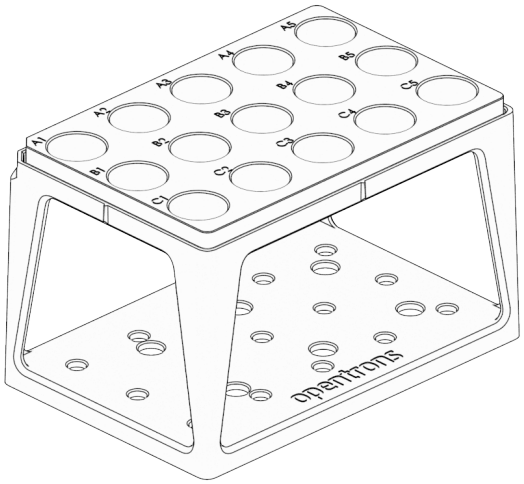
This tuberack insert definition is for screwcap tubes ranging from 2-5ml.



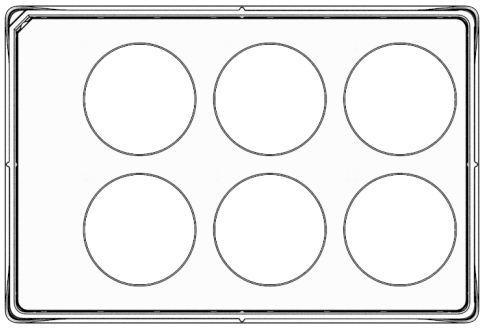

opentrons

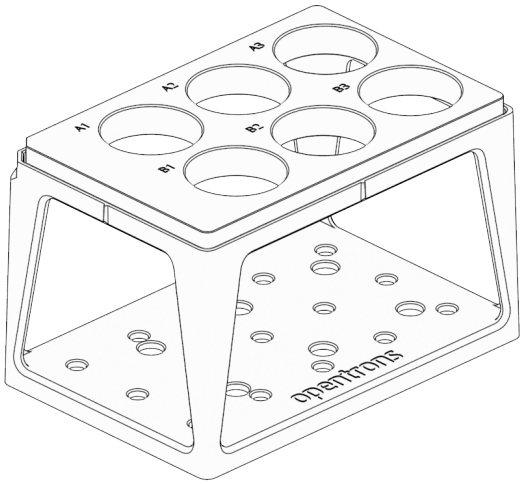
opentrons-tuberack-15ml



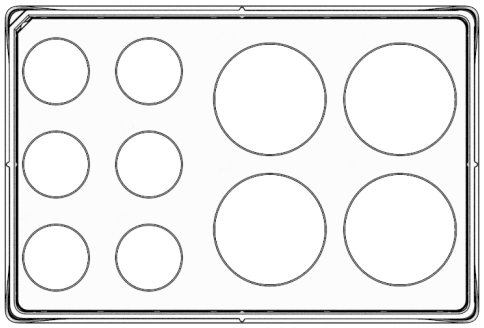


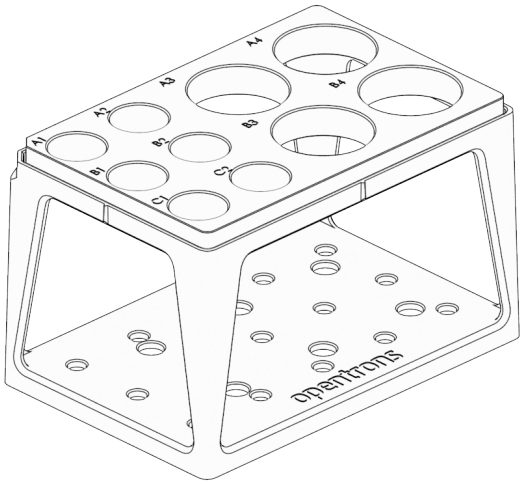
opentrons-tuberack-50ml





opentrons-tuberack-15_50ml





Point

Use `point` when there is only one position per labware, such as a scale.

```
my_container = labware.load('point', slot)
```

You can access the point position as `my_labware.wells('A1')` or `my_labware.wells(0)`.

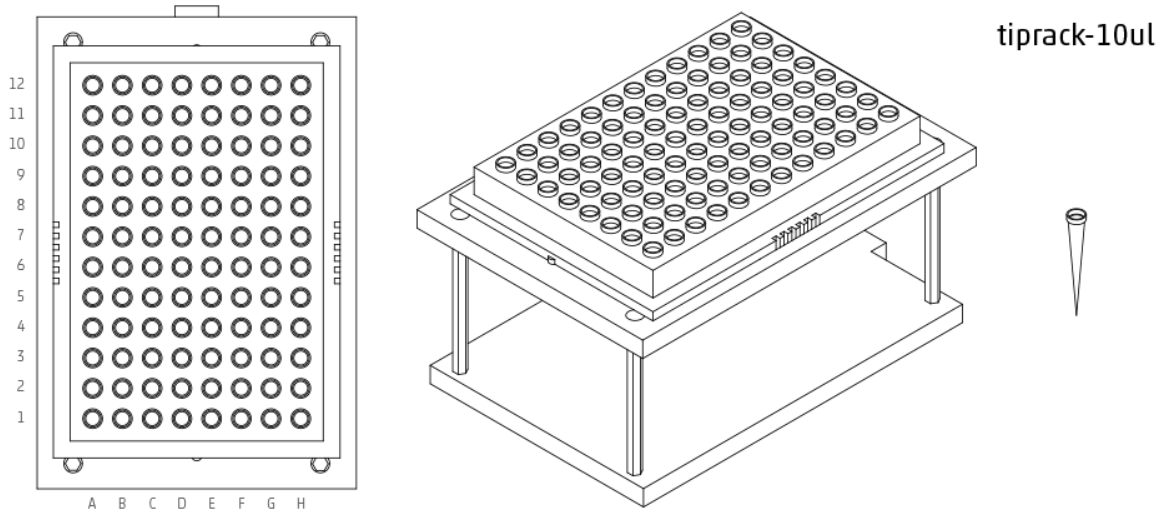
Tipracks

tiprack-10ul

Tip rack for a 10 uL pipette (single or 8-channel)

```
labware.load('tiprack-10ul', slot)
```

Accessing Tips: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

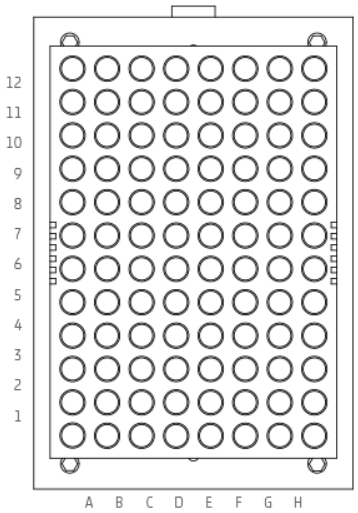


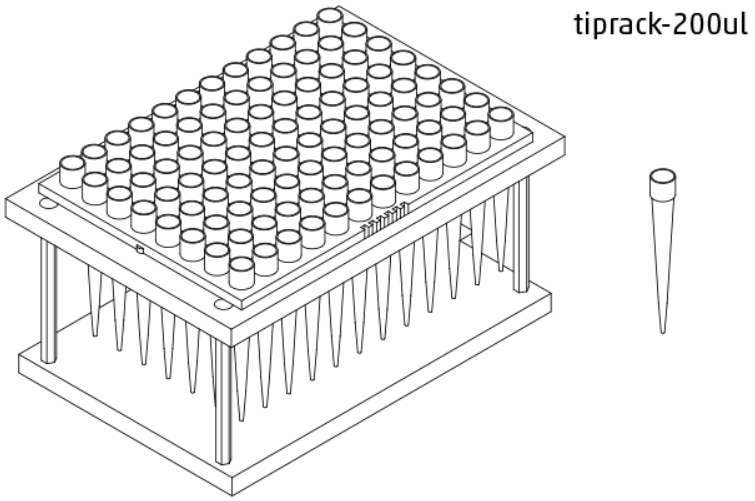
tiprack-200ul

Tip rack for a 200 or 300 uL pipette (single or 8-channel)

```
labware.load('tiprack-200ul', slot)
```

Accessing Tips: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`





opentrons-tiprack-300ul

This is a custom-made 300ul tiprack for the OT 2 model (single or 8-channel)

```
labware.load('opentrons-tiprack-300ul', slot)
```

Accessing Tips: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

Check out our available tipracks here[18]

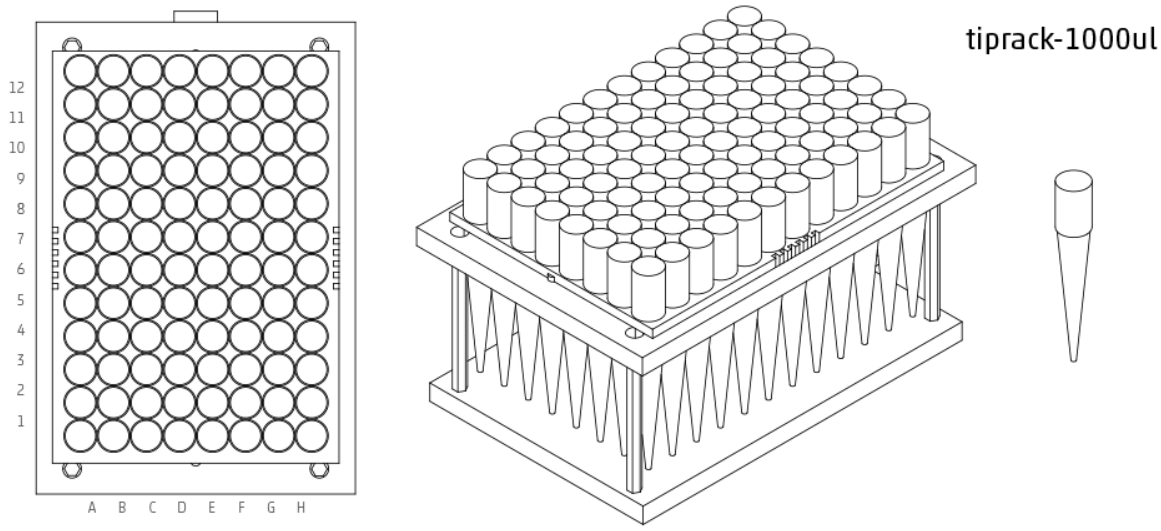
tiprack-1000ul

Tip rack for a 1000 uL pipette (single or 8-channel)

```
labware.load('tiprack-1000ul', slot)
```

Accessing Tips: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

[18] https://shop.opentrons.com/collections/opentrons-tips

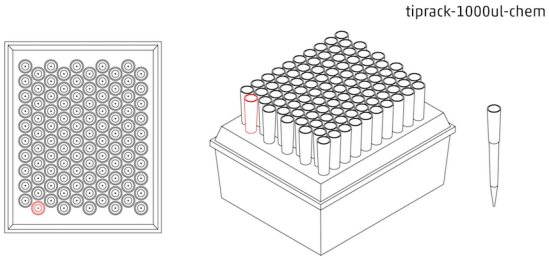


tiprack-1000ul-chem

Tip rack for 1000ul chem (10x10)

```
labware.load('tiprack-1000ul-chem', slot)
```

Accessing Tips: *single channel* `[0]-[99]`



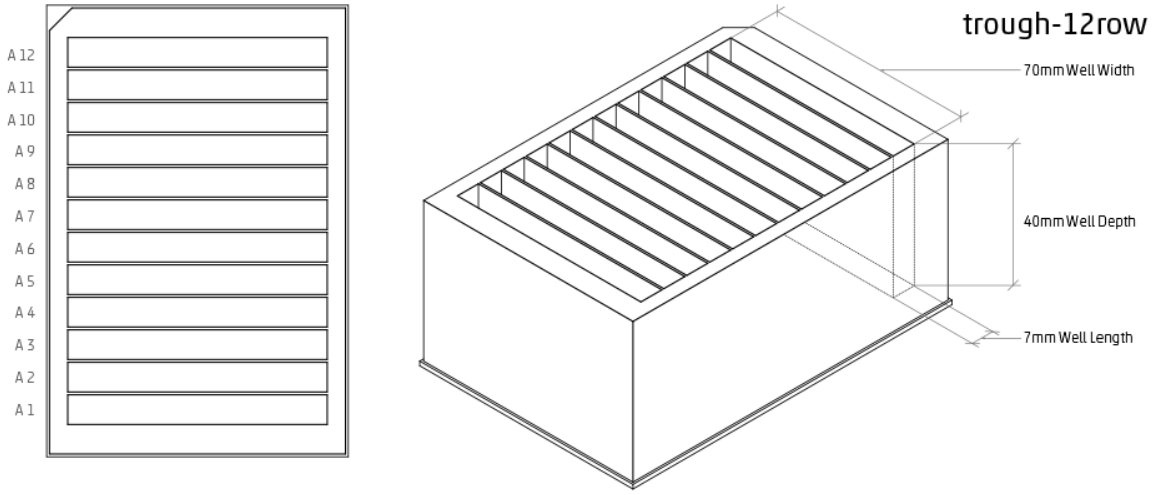
Troughs

trough-12row

12 row reservoir

```
labware.load('trough-12row', slot)
```

Accessing Rows: *single channel* or *8-channel* `['A1']-['A12']`



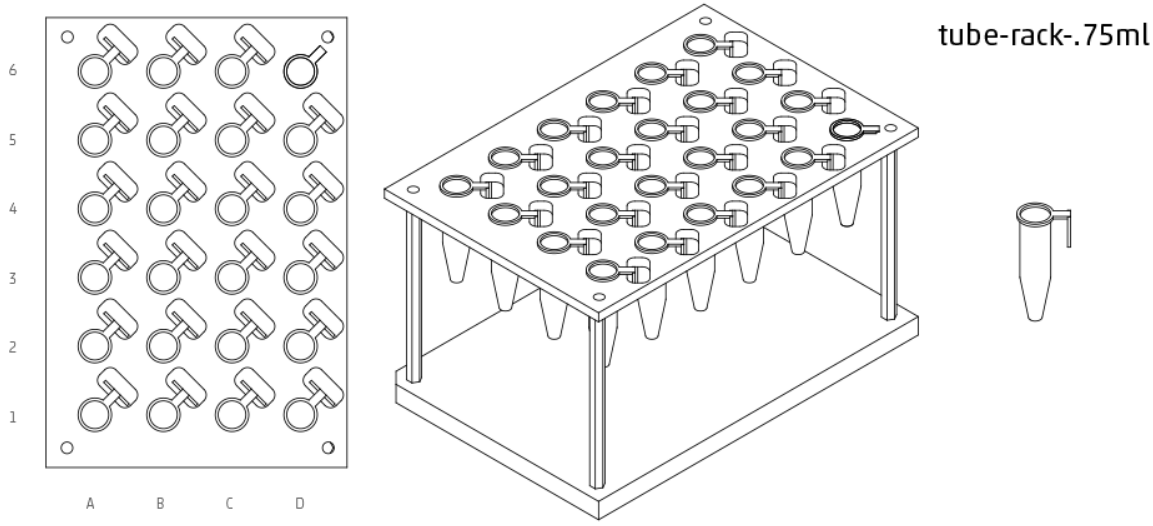
Tube Racks

tube-rack-.75ml

4x6 rack that holds .75 mL microcentrifuge tubes

```
labware.load('tube-rack-.75ml', slot)
```

Accessing Tubes: *single channel* `['A1']-['D6']`

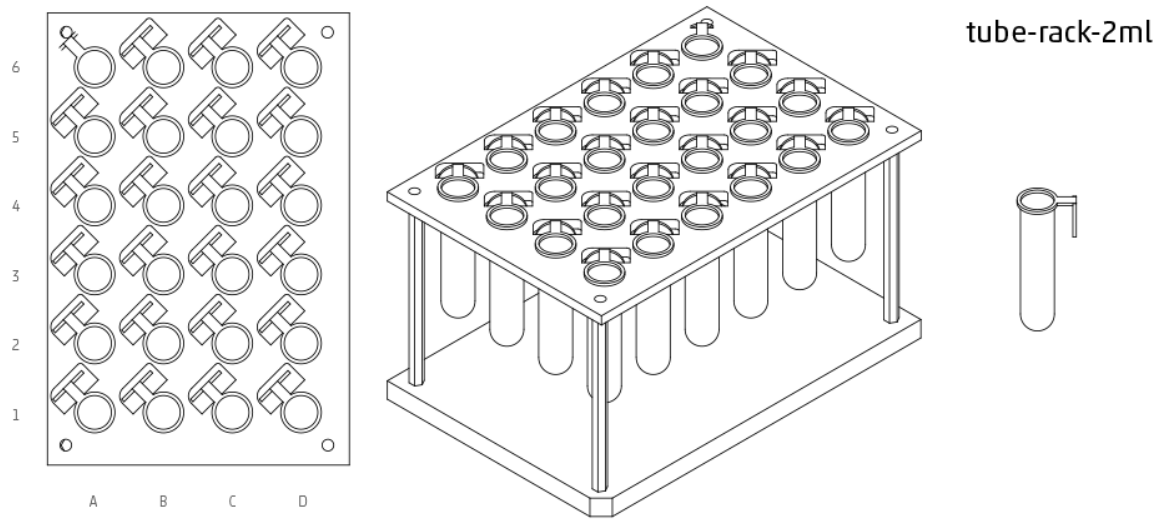


tube-rack-2ml

4x6 rack that holds 1.5 mL microcentrifuge tubes and 2 mL microcentrifuge tubes

```
labware.load('tube-rack-2ml', slot)
```

Accessing Tubes: *single channel* `['A1']-['D6']`

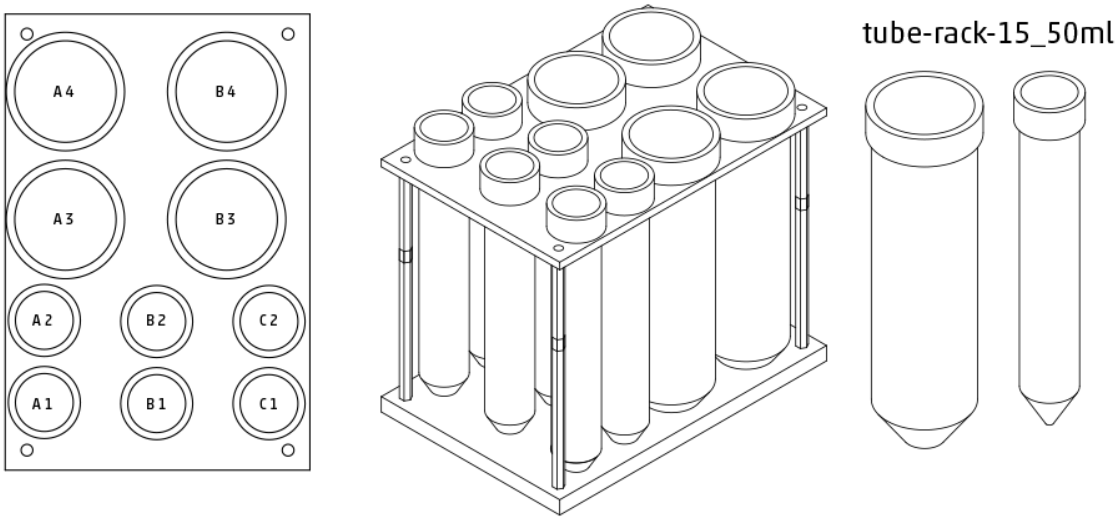


tube-rack-15_50ml

rack that holds 6 15 mL tubes and 4 50 mL tubes

```
labware.load('tube-rack-15_50ml', slot)
```

Accessing Tubes: *single channel* `['A1']-['A3'],['B1']-['B3'],['C1']-['C2'],['D1']-['D2']`



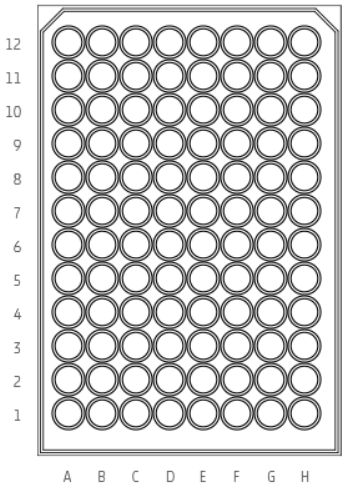
Plates

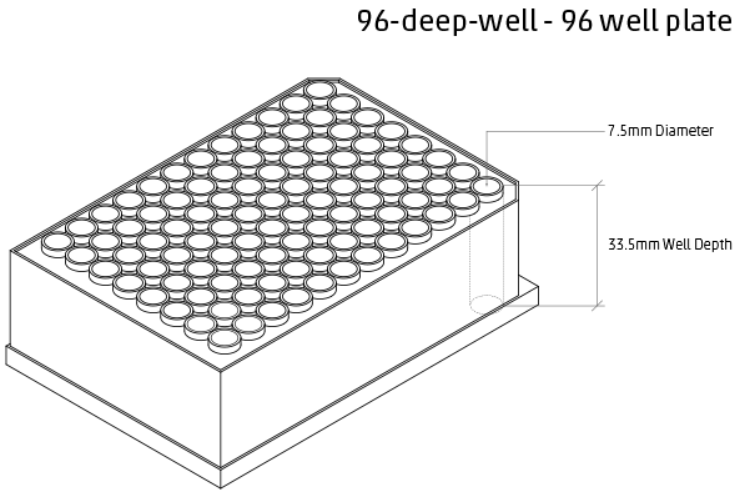
96-deep-well

See dimensions in diagram below.

```
labware.load('96-deep-well', slot)
```

Accessing Wells: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`



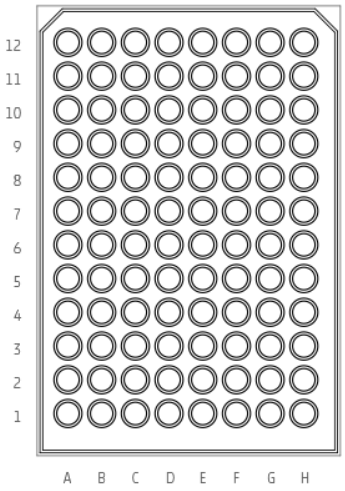


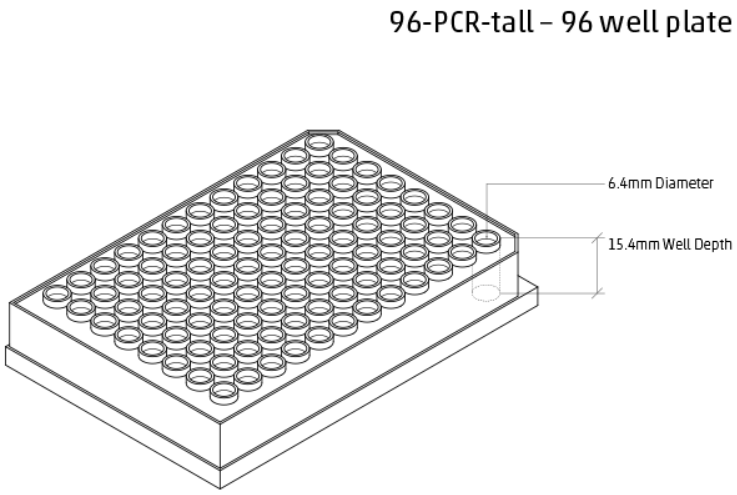
96-PCR-tall

See dimensions in diagram below.

```
labware.load('96-PCR-tall', slot)
```

Accessing Wells: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`



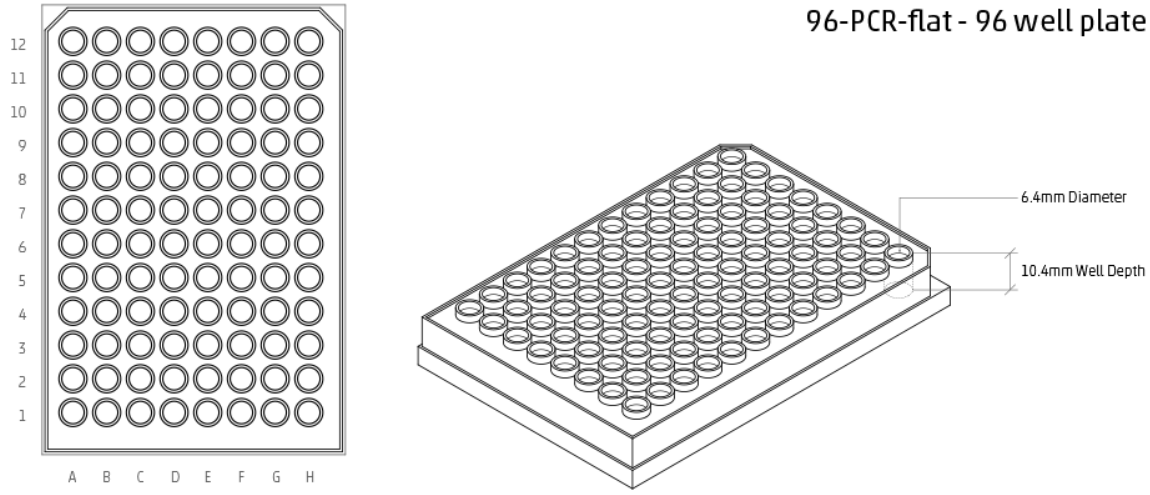


96-PCR-flat

See dimensions in diagram below.

```
labware.load('96-PCR-flat', slot)
```

Accessing Wells: *single channel* `['A1']-['H12']`, *8-channel* `['A1']-['A12']`

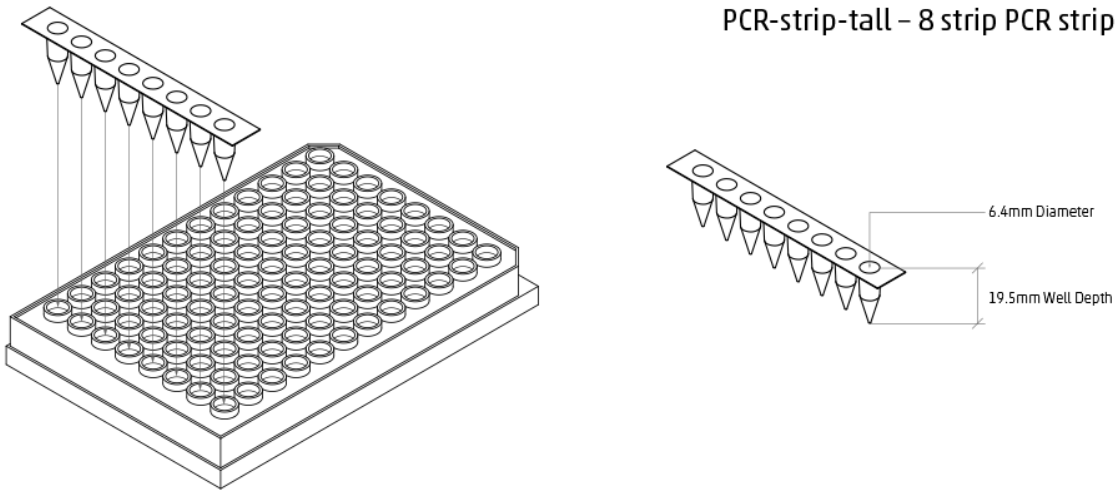


PCR-strip-tall

See dimensions in diagram below.

```
labware.load('PCR-strip-tall', slot)
```

Accessing Wells: *single channel* `['A1']-['A8']`, *8-channel* `['A1']`

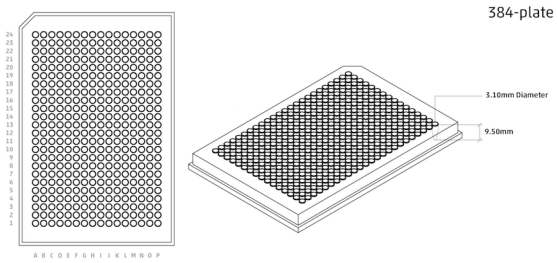


384-plate

See dimensions in diagram below.

```
labware.load('384-plate', slot)
```

Accessing Wells: *single channel* `['A1']-['P24']`, *multi-channel* `['A1']-['A24]`



Labware Module

```
'''
Examples in this section require the following
'''
from opentrons import labware
```

List

Once the labware module is loaded, you can see a list of all containers currently inside the API by calling `labware.list()`

```
labware.list()
```

Load

Labware is loaded with two arguments: 1) the labware type, and 2) the deck slot it will be placed in on the robot.

```
p = labware.load('96-flat', '1')
```

A third optional argument can be used to give a labware a unique name.

```
p = labware.load('96-flat', '2', 'any-name-you-want')
```

Unique names are useful in a few scenarios. First, they allow the labware to have independent calibration data from other labware in the same slot. In the example above, the container named 'any-name-you-want' will assume different calibration data from the unnamed plate, even though they are the same type and in the same slot.

Note: Calibration data refers to the saved positions for each labware on deck, and is a part of the Opentrons App calibration procedure[19].

---

[19] https://support.opentrons.com/guide-for-getting-started-with-the-ot-2/6-calibrate-the-ot-2/b-pipette-and-labware-calibration

Create

In addition to the default labware that come with the Opentrons API, you can create your own custom labware.

Through the API's call labware.create(), you can create simple grid labware, which consist of circular wells arranged in columns and rows.

```
plate_name = '3x6_plate'
if plate_name not in labware.list():
    custom_plate = labware.create(
        plate_name,                    # name of you labware
        grid=(3, 6),                    # specify amount of (columns, rows)
        spacing=(12, 12),               # distances (mm) between each (column, row)
        diameter=5,                     # diameter (mm) of each well on the plate
        depth=10,                       # depth (mm) of each well on the plate
        volume=200)
```

When you create your custom labware it will return the custom plate. You should only need to run this once among all of your protocols for the same custom labware because the data is automatically saved on the robot.

In this example, the call to *labware.create* is wrapped in an if-block, so that it does not try to add the definition to a robot where this has already been run (which would cause an error). If a labware has already been added to the database (by previously calling *labware.create*, the if-block will not execute, and the rest of the protocol will use the definition that was already created and calibrated.

Note There is some specialty labware that will require you to specify the type within your labware name. If you are creating a custom tiprack, it must be *tiprack*-REST-OF-LABWARE-NAME in order for the program to act reliably.

If you would like to delete a labware you have already added to the database, you can do the following:

```
from opentrons.data_storage import database
database.delete_container('3x6_plate')
```

This allows you to make changes to the labware within the database under the same name.

```
for well in custom_plate.wells():
    print(well)
```

will print out...

```
<Well A1>
<Well B1>
<Well C1>
<Well A2>
<Well B2>
<Well C2>
<Well A3>
<Well B3>
<Well C3>
<Well A4>
<Well B4>
<Well C4>
<Well A5>
<Well B5>
<Well C5>
<Well A6>
<Well B6>
<Well C6>
```

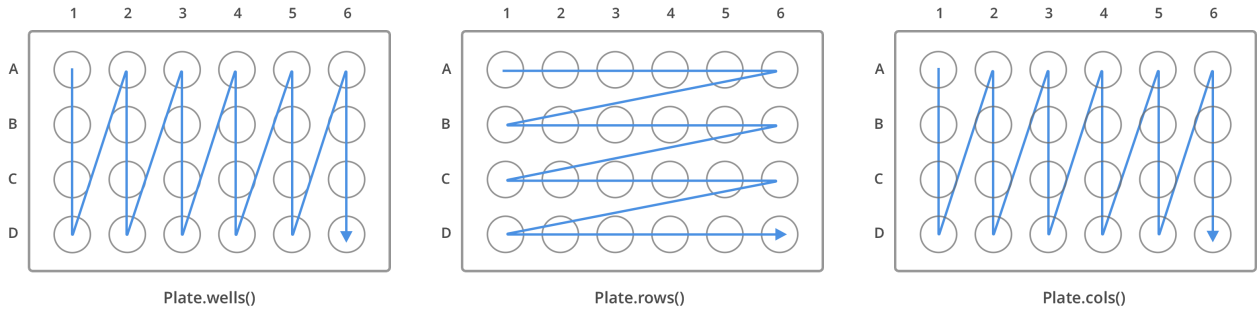
```
from opentrons import labware, robot

plate = labware.load('96-flat', 'A1')
```

Accessing Wells

Individual Wells

When writing a protocol using the API, you will be spending most of your time selecting which wells to transfer liquids to and from.

The OT-One deck and labware are all set up with the same coordinate system - lettered rows `['A']-['END']` and numbered columns `['1']-['END']`.




```
'''
Examples in this section expect the following
'''
from opentrons import labware

plate = labware.load('96-flat', '1')
```

Wells by Name

Once a labware is loaded into your protocol, you can easily access the many wells within it using `wells()` method. `wells()` takes the name of the well as an argument, and will return the well at that location.

```
plate.wells('A1')
plate.wells('H12')
```

Wells by Index

Wells can be referenced by their "string" name, as demonstrated above. However, they can also be referenced with zero-indexing, with the first well in a labware being at position 0.

```
plate.wells(0)    # well A1
plate.wells(95)   # well H12
```

Columns and Rows

A labware's wells are organized within a series of columns and rows, which are also labelled on standard labware. In the API, rows are given letter names (`'A'` through `'H'` for example) and go left to right, while columns are given numbered names (`'1'` through `'12'` for example) and go from front to back. You can access a specific row or column by using the `rows()` and `columns()` methods on a labware. These will return all wells within that row or column.

```
row = plate.rows('A')
column = plate.columns('1')

print('Column "1" has', len(column), 'wells')
print('Row "A" has', len(row), 'wells')
```

will print out...

```
Column "1" has 8 wells
Row "A" has 12 wells
```

The `rows()` or `cols()` methods can be used in combination with the `wells()` method to access wells within that row or column. In the example below, both lines refer to well `'A1'`.

```
plate.cols('1').wells('A')
plate.rows('A').wells('1')
```

```
from opentrons import labware, robot

plate = labware.load('96-flat', '1')
```

Multiple Wells

If we had to reference each well one at a time, our protocols could get very very long.

When describing a liquid transfer, we can point to groups of wells for the liquid's source and/or destination. Or, we can get a group of wells that we want to loop through.

```
'''
Examples in this section expect the following
'''
from opentrons import labware

plate = labware.load('96-flat', '2')
```

Wells

The `wells()` method can return a single well, or it can return a list of wells when multiple arguments are passed.

Here is an example or accessing a list of wells, each specified by name:

```
w = plate.wells('A1', 'B2', 'C3', 'H12')

print(w)
```

will print out...

```
<WellSeries: <Well A1><Well B2><Well C3><Well H12>>
```

Multiple wells can be treated just like a normal Python list, and can be iterated through:

```
for w in plate.wells('A1', 'B2', 'C3', 'H12'):
    print(w)
```

will print out...

```
<Well A1>
<Well B2>
<Well C3>
<Well H12>
```

Wells To

Instead of having to list the name of every well, we can also create a range of wells with a start and end point. The first argument is the starting well, and the `to=` argument is the last well.

```
for w in plate.wells('A1', to='H1'):
    print(w)
```

will print out...

```
<Well A1>
<Well B1>
<Well C1>
<Well D1>
<Well E1>
<Well F1>
<Well G1>
<Well H1>
```

These lists of wells can also move in the reverse direction along your labware. For example, setting the `to=` argument to a well that comes before the starting position is allowed:

```
for w in plate.wells('H1', to='A1'):
    print(w)
```

will print out...

```
<Well H1>
<Well G1>
<Well F1>
<Well E1>
<Well C1>
<Well B1>
<Well A1>
```

Wells Length

Another way you can create a list of wells is by specifying the length= of the well list you need, in addition to the starting point. The example below will return eight wells, starting at well `'A1'`:

```
for w in plate.wells('A1', length=8):
    print(w)
```

will print out...

```
<Well A1>
<Well B1>
<Well C1>
<Well D1>
<Well E1>
<Well F1>
<Well G1>
<Well H1>
```

Columns and Rows

Columns and Rows The same arguments described above can be used with `rows()` and `cols()` to create lists of rows or columns.

Here is an example of iterating through rows:

```
for r in plate.rows('A', length=3):
    print(r)
```

will print out...

```

```

<WellSeries: <WellSeries: <Well A1><Well A2><Well A3><Well A4><Well A5><Well A6><Well A7><Well A8><Well A9><Well A10><Well A11><Well A12>> <WellSeries: <Well B1><Well B2><Well B3><Well B4><Well B5><Well B6><Well B7><Well B8><Well B9><Well B10><Well B11><Well B12>> <WellSeries: <Well C1><Well C2><Well C3><Well C4><Well C5><Well C6><Well C7><Well C8><Well C9><Well C10><Well C11><Well C12>>

>

And here is an example of iterating through columns:

```
for c in plate.cols('1', to='10'):
    print(c)
```

will print out...

```

```

<WellSeries: <WellSeries: <Well A1><Well B1><Well C1><Well D1><Well E1><Well F1><Well G1><Well H1>> <WellSeries: <Well A2><Well B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>> <WellSeries: <Well A3><Well B3><Well C3><Well D3><Well E3><Well F3><Well G3><Well H3>> <WellSeries: <Well A4><Well B4><Well C4><Well D4><Well E4><Well F4><Well G4><Well H4>> <WellSeries: <Well A5><Well B5><Well C5><Well D5><Well E5><Well F5><Well G5><Well H5>> <WellSeries: <Well A6><Well B6><Well C6><Well D6><Well E6><Well F6><Well G6><Well H6>> <WellSeries: <Well A7><Well B7><Well C7><Well D7><Well E7><Well F7><Well G7><Well H7>> <WellSeries: <Well A8><Well B8><Well C8><Well D8><Well E8><Well F8><Well G8><Well H8>> <WellSeries: <Well A9><Well B9><Well C9><Well D9><Well E9><Well F9><Well G9><Well H9>> <WellSeries: <Well A10><Well B10><Well C10><Well D10><Well E10><Well F10><Well G10><Well H10>>

>

Slices

Labware can also be treating similarly to Python lists, and can therefore handle slices.

```
for w in plate[0:8:2]:
    print(w)
```

will print out...

```
<Well A1>
<Well C1>
<Well E1>
<Well G1>
```

The API's labware are also prepared to take string values for the slice's `start` and `stop` positions.

```
for w in plate['A1':'A2':2]:
    print(w)
```

will print out...

```
<Well A1>
<Well C1>
<Well E1>
<Well G1>
```

```
for w in plate.rows['B']['1'::2]:
    print(w)
```

will print out...

```
<Well B1>
<Well B3>
<Well B5>
<Well B7>
<Well B9>
<Well B11>
```

The `instruments` module gives your protocol access to the pipette constructors, which is what you will be primarily using to create protocol commands.

Creating a Pipette

```
'''
Examples in this section require the following:
'''
from opentrons import instruments, robot
```

Pipette Model(s)

Currently in our API there are 7 pipette models to correspond with the offered pipette models on our website.

They are as follows: `P10_Single` (1 - 10 ul) `P10_Multi` (1 - 10ul) `P50_Single` (5 - 50ul) `P50_Multi` (5 - 50ul) `P300_Single` (30 - 300ul) `P300_Mutli` (30 - 300ul) `P1000_Single` (100 - 1000ul)

For every pipette type you are using in a protocol, you must use one of the model names specified above and call it out as `instruments`.(Model Name)

Mount

To create a pipette object, you must give it a mount. The mount can be either `'left'` or `'right'`. In this example, we are using a Single-Channel 300uL pipette.

```
pipette = instruments.P300_Single(mount='left')
```

Plunger Flow Rates

The speeds at which the pipette will aspirate and dispense can be set through `aspirate_speed` and `dispense_speed`. The values are in microliters/seconds, and have varying defaults depending on the model.

```
pipette = instruments.P300_Single(
    mount='right',
    aspirate_flow_rate=200,
    dispense_flow_rate=600)
```

Minimum and Maximum Volume

The minimum and maximum volume of the pipette may be set using `min_volume` and `max_volume`. The values are in microliters and have varying defaults depending on the model.

```
pipette = instruments.P10_Single(
    mount='right',
    min_volume=2,
    max_volume=8)
```

The given defaults for every pipette model is the following:

P10_Single

- Aspirate Default: 5 μl/s
- Dispense Default: 10 μl/s
- Minimum Volume: 1 μl
- Maximum Volume: 10 μl

P10_Multi

- Aspirate Default: 5 µl/s
- Dispense Default: 10 µl/s
- Minimum Volume: 1 µl
- Maximum Volume: 10 µl

P50_Single

- Aspirate Default: 25 µl/s
- Dispense Default: 50 µl/s
- Minimum Volume: 5 µl
- Maximum Volume: 50 µl

P50_Multi

- Aspirate Default: 25 µl/s
- Dispense Default: 50 µl/s
- Minimum Volume: 5 µl
- Maximum Volume: 50 µl

P300_Single

- Aspirate Default: 150 µl/s
- Dispense Default: 300 µl/s
- Minimum Volume: 30 µl
- Maximum Volume: 300 µl

P300_Multi

- Aspirate Default: 150 µl/s
- Dispense Default: 300 µl/s
- Minimum Volume: 30 µl
- Maximum Volume: 300 µl

P1000_Single

- Aspirate Default: 500 µl/s
- Dispense Default: 1000 µl/s
- Minimum Volume: 100 µl

- Maximum Volume: 1000 μl

Old Pipette Constructor

The `Pipette` constructor that was used directly in OT-One protocols is now an internal-only class. Its behavior is difficult to predict when not used through the public constructors mentioned above. `Pipette` constructor arguments are subject to change of their default values, behaviors, and parameters may be added or removed without warning or a major version increment.

Atomic Liquid Handling

Tip Handling

When we handle liquids with a pipette, we are constantly exchanging old, used tips for new ones to prevent cross-contamination between our wells. To help with this constant need, we describe in this section a few methods for getting new tips, and removing tips from a pipette.

This section demonstrates the options available for controlling tips

```
'''
Examples in this section expect the following
'''
from opentrons import labware, instruments, robot

tiprack = labware.load('tiprack-200ul', '2')

pipette = instruments.P300_Single(mount='left')
```

Pick Up Tip

Before any liquid handling can be done, your pipette must have a tip on it. The command `pick_up_tip()` will move the pipette over to the specified tip, the press down into it to create a vacuum seal. The below example picks up the tip at location `'A1'`.

```
pipette.pick_up_tip(tiprack.wells('A1'))
```

Drop Tip

Once finished with a tip, the pipette will autonomously remove the tip when we call `drop_tip()`. We can specify where to drop the tip by passing in a location. The below example drops the tip back at its originating location on the tip rack. If no location is specified, it will go to the fixed trash location on the deck. .. code-block:: python

    pipette.drop_tip(tiprack.wells('A1'))

Instead of returning a tip to the tip rack, we can also drop it in an alternative trash container besides the fixed trash on the deck.

```
trash = labware.load('trash-box', '1')
pipette.pick_up_tip(tiprack.wells('A2'))
pipette.drop_tip(trash)
```

Return Tip

When we need to return the tip to its originating location on the tip rack, we can simply call `return_tip()`. The example below will automatically return the tip to `'A3'` on the tip rack.

```
pipette.pick_up_tip(tiprack.wells('A3'))
pipette.return_tip()
```

Tips Iterating

Automatically iterate through tips and drop tip in trash by attaching containers to a pipette. If no location is specified, the pipette will move to the next available tip by iterating through the tiprack that is associated with it.

```
'''
Examples in this section expect the following
'''
from opentrons import labware, instruments, robot

trash = labware.load('trash-box', '1')
tip_rack_1 = containers.load('tiprack-200ul', '2')
tip_rack_2 = containers.load('tiprack-200ul', '3')
```

Attach Tip Rack to Pipette

Tip racks and trash containers can be "attached" to a pipette when the pipette is created. This give the pipette the ability to automatically iterate through tips, and to automatically send the tip to the trash container.

Trash containers can be attached with the option `trash_container=TRASH_CONTAINER`.

Multiple tip racks are can be attached with the option `tip_racks=[RACK_1,RACK_2,etc...  ]`.

```
pipette = instruments.P300_Single(mount='left',
                                  tip_racks=[tip_rack_1, tip_rack_2],
                                  trash_container=trash)
```

Note: The `tip_racks=` option expects us to give it a Python list, containing each tip rack we want to attach. If we are only attaching one tip rack, then the list will have a length of one, like the following:

`tip_racks=[tiprack]`

Iterating Through Tips

Now that we have two tip racks attached to the pipette, we can automatically step through each tip whenever we call `pick_up_tip()`. We then have the option to either `return_tip()` to the tip rack, or we can `drop_tip()` to remove the tip in the attached trash container.

```
pipette.pick_up_tip()  # picks up tip_rack_1:A1
pipette.return_tip()
pipette.pick_up_tip()  # picks up tip_rack_1:A2
```

```
pipette.drop_tip()      # automatically drops in trash

# use loop to pick up tips tip_rack_1:A3 through tip_rack_2:H12
tips_left = 94 + 96 # add up the number of tips leftover in both tipracks
for _ in range(tips_left):
    pipette.pick_up_tip()
    pipette.return_tip()
```

If we try to `pick_up_tip()` again when all the tips have been used, the Opentrons API will show you an error.

Note: If you run the cell above, and then uncomment and run the cell below, you will get an error because the pipette is out of tips.

```
# this will raise an exception if run after the previous code block
# pipette.pick_up_tip()
```

Reseting Tip Tracking

If you plan to change out tipracks during the protocol run, you must reset tip tracking to prevent any errors. This is done through `pipette.reset()` which resets the tipracks and sets the current volume back to 0 ul.

```
pipette.reset()
```

Select Starting Tip

Calls to `pick_up_tip()` will by default start at the attached tip rack's `'A1'` location in order of tipracks listed. If you however want to start automatic tip iterating at a different tip, you can use `start_at_tip()`.

```
pipette.start_at_tip(tip_rack_1.well('C3'))
pipette.pick_up_tip()  # pick up C3 from "tip_rack_1"
pipette.return_tip()
```

Get Current Tip

Get the source location of the pipette's current tip by calling `current_tip()`. If the tip was from the `'A1'` position on our tip rack, `current_tip()` will return that position.

```
print(pipette.current_tip())  # is holding no tip

pipette.pick_up_tip()
print(pipette.current_tip())  # is holding the next available tip

pipette.return_tip()
print(pipette.current_tip())  # is holding no tip
```

will print out...

Liquid Control

This is the fun section, where we get to move things around and pipette! This section describes the `Pipette` object's many liquid-handling commands, as well as how to move the `robot`. Please note that the default now for pipette aspirate and dispense location is a 1mm offset from the **bottom** of the well now.

```
from opentrons import labware, instruments, robot

'''
Examples in this section expect the following:
'''
plate = labware.load('96-flat', '1')
pipette = instruments.P300_Single(mount='left')
pipette.pick_up_tip()
```

Aspirate

To aspirate is to pull liquid up into the pipette's tip. When calling aspirate on a pipette, we can specify how many micoliters, and at which location, to draw liquid from:

```
pipette.aspirate(50, plate.wells('A1'))  # aspirate 50uL from plate:A1
```

Now our pipette's tip is holding 50uL.

We can also simply specify how many microliters to aspirate, and not mention a location. The pipette in this circumstance will aspirate from it's current location (which we previously set as `plate.wells('A1')`.

```
pipette.aspirate(50)                     # aspirate 50uL from current position
```

Now our pipette's tip is holding 100uL.

We can also specify only the location to aspirate from. If we do not tell the pipette how many micoliters to aspirate, it will by default fill up the remaining volume in it's tip. In this example, since we already have 100uL in the tip, the pipette will aspirate another 200uL

```
pipette.aspirate(plate.wells('A2'))      # aspirate until pipette fills from plate:A2
```

Dispense

To dispense is to push out liquid from the pipette's tip. It's usage in the Opentrons API is nearly identical to `aspirate()`, in that you can specify microliters and location, only microliters, or only a location:

```
pipette.dispense(50, plate.wells('B1')) # dispense 50uL to plate:B1
pipette.dispense(50)                    # dispense 50uL to current position
pipette.dispense(plate.wells('B2'))     # dispense until pipette empties to plate:B2
```

That final dispense without specifying a micoliter amount will dispense all remaining liquids in the tip to `plate.wells('B2')`, and now our pipette is empty.

Blow Out

To blow out is to push an extra amount of air through the pipette's tip, so as to make sure that any remaining droplets are expelled.

When calling `blow_out()` on a pipette, we have the option to specify a location to blow out the remaining liquid. If no location is specified, the pipette will blow out from it's current position.

```
pipette.blow_out()                    # blow out in current location
pipette.blow_out(plate.wells('B3')) # blow out in current plate:B3
```

Touch Tip

To touch tip is to move the pipette's currently attached tip to four opposite edges of a well, for the purpose of knocking off any droplets that might be hanging from the tip.

When calling `touch_tip()` on a pipette, we have the option to specify a location where the tip will touch the inner walls. If no location is specified, the pipette will touch tip inside it's current location.

```
pipette.touch_tip()                    # touch tip within current location
pipette.touch_tip(-2)                  # touch tip 2mm below the top of the current␣
↪location
pipette.touch_tip(plate.wells('B1')) # touch tip within plate:B1
```

Mix

Mixing is simply performing a series of `aspirate()` and `dispense()` commands in a row on a single location. However, instead of having to write those commands out every time, the Opentrons API allows you to simply say `mix()`.

The mix command takes three arguments: `mix(repetitions,volume,location)`

```
pipette.mix(4, 100, plate.wells('A2'))    # mix 4 times, 100uL, in plate:A2
pipette.mix(3, 50)                        # mix 3 times, 50uL, in current location
pipette.mix(2)                            # mix 2 times, pipette's max volume, in␣
↪current location
```

Air Gap

Some liquids need an extra amount of air in the pipette's tip to prevent it from sliding out. A call to `air_gap()` with a microliter amount will aspirate that much air into the tip.

```
pipette.aspirate(100, plate.wells('B4'))
pipette.air_gap(20)
pipette.drop_tip()
```

```
from opentrons import labware, instruments, robot

'''
Examples in this section expect the following
'''
```

```
tiprack = labware.load('tiprack-200ul', '1')
plate = labware.load('96-flat', '2')

pipette = instruments.P300_Single(mount='right', tip_racks=[tiprack])
```

Controlling Speed

You can change the speed at which you aspirate or dispense liquid by either changing the defaults in the pipette constructor (more info under the *Creating a Pipette* section) or using our *set_flow_rate* function. This can be called at any time during the protocol.

```
from opentrons import labware, instruments, robot

'''
Examples in this section expect the following
'''
tiprack = labware.load('tiprack-200ul', '1')
plate = labware.load('96-flat', '2')

pipette = instruments.P300_Single(mount='right', tip_racks=[tiprack])

pipette.set_flow_rate(aspirate=50, dispense=100)
```

You can also choose to only update aspirate OR dispense depending on the application. Pipette liquid handling speed is in *ul/s*.

Note The dispense speed also controls the speed of *blow_out*.

Moving

Move To

Pipette's are able to `move_to()` any location on the deck.

For example, we can move to the first tip in our tip rack:

```
pipette.move_to(tiprack.wells('A1'))
```

You can also specify at what height you would like the robot to move to inside of a location using `top()` and `bottom()` methods on that location.

```
pipette.move_to(plate.wells('A1').bottom())   # move to the bottom of well A1
pipette.move_to(plate.wells('A1').top())      # move to the top of well A1
pipette.move_to(plate.wells('A1').bottom(2)) # move to 2mm above the bottom of well A1
pipette.move_to(plate.wells('A1').top(-2))    # move to 2mm below the top of well A1
```

The above commands will cause the robot's head to first move upwards, then over to above the target location, then finally downwards until the target location is reached. If instead you would like the robot to mive in a straight line to the target location, you can set the movement strategy to `'direct'`.

```
pipette.move_to(plate.wells('A1'), strategy='direct')
```

Note: Moving with `strategy='direct'` will run the risk of colliding with things on your deck. Be very careful when using this option.

Usually the `strategy='direct'` option is useful when moving inside of a well. Take a look at the below sequence of movements, which first move the head to a well, and use 'direct' movements inside that well, then finally move on to a different well.

```
pipette.move_to(plate.wells('A1'))
pipette.move_to(plate.wells('A1').bottom(1), strategy='direct')
pipette.move_to(plate.wells('A1').top(-2), strategy='direct')
pipette.move_to(plate.wells('A1'))
```

Delay

To have your protocol pause for any given number of minutes or seconds, simply call `delay()` on your pipette. The value passed into `delay()` is the number of minutes or seconds the robot will wait until moving on to the next commands.

```
pipette.delay(seconds=2)              # pause for 2 seconds
pipette.delay(minutes=5)              # pause for 5 minutes
pipette.delay(minutes=5, seconds=2)   # pause for 5 minutes and 2 seconds
```

Complex Liquid Handling

The examples below will use the following set-up:

```
from opentrons import robot, labware, instruments

robot.clear_commands()

plate = labware.load('96-flat', '1')

tiprack = labware.load('tiprack-200ul', '2')

pipette = instruments.P300_Single(
    mount='left',
    tip_racks=[tiprack])
```

Transfer

Most of time, a protocol is really just looping over some wells, aspirating, and then dispensing. Even though they are simple in nature, these loops take up a lot of space. The `pipette.transfer()` command takes care of those common loops. It will combine aspirates and dispenses automatically, making your protocol easier to read and edit.

Basic

The example below will transfer 100 uL from well `'A1'` to well `'B1'`, automatically picking up a new tip and then disposing it when finished.

```
pipette.transfer(100, plate.wells('A1'), plate.wells('B1'))
```

Transfer commands will automatically create entire series of `aspirate()`, `dispense()`, and other `Pipette` commands.

Large Volumes

Volumes larger than the pipette's `max_volume` will automatically divide into smaller transfers.

```
pipette.transfer(700, plate.wells('A2'), plate.wells('B2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 700 from <Well A2> to <Well B2>
Picking up tip <Well A1>
Aspirating 200.0 uL from <Well A2> at 1 speed
Dispensing 200.0 uL into <Well B2>
Aspirating 200.0 uL from <Well A2> at 1 speed
Dispensing 200.0 uL into <Well B2>
Aspirating 150.0 uL from <Well A2> at 1 speed
Dispensing 150.0 uL into <Well B2>
Aspirating 150.0 uL from <Well A2> at 1 speed
Dispensing 150.0 uL into <Well B2>
Dropping tip <Well A1>
```

Multiple Wells

Transfer commands are most useful when moving liquid between multiple wells.

```
pipette.transfer(100, plate.cols('1'), plate.cols('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <WellSeries: <Well A1><Well A2><Well A3><Well A4><Well A5><Well␣
↪A6><Well A7><Well A8><Well A9><Well A10><Well A11><Well A12>> to <WellSeries: <Well␣
↪B1><Well B2><Well B3><Well B4><Well B5><Well B6><Well B7><Well B8><Well B9><Well␣
↪B10><Well B11><Well B12>>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B1>
Aspirating 100.0 uL from <Well A2> at 1 speed
Dispensing 100.0 uL into <Well B2>
Aspirating 100.0 uL from <Well A3> at 1 speed
Dispensing 100.0 uL into <Well B3>
Aspirating 100.0 uL from <Well A4> at 1 speed
Dispensing 100.0 uL into <Well B4>
Aspirating 100.0 uL from <Well A5> at 1 speed
Dispensing 100.0 uL into <Well B5>
```

```
Aspirating 100.0 uL from <Well A6> at 1 speed
Dispensing 100.0 uL into <Well B6>
Aspirating 100.0 uL from <Well A7> at 1 speed
Dispensing 100.0 uL into <Well B7>
Aspirating 100.0 uL from <Well A8> at 1 speed
Dispensing 100.0 uL into <Well B8>
Aspirating 100.0 uL from <Well A9> at 1 speed
Dispensing 100.0 uL into <Well B9>
Aspirating 100.0 uL from <Well A10> at 1 speed
Dispensing 100.0 uL into <Well B10>
Aspirating 100.0 uL from <Well A11> at 1 speed
Dispensing 100.0 uL into <Well B11>
Aspirating 100.0 uL from <Well A12> at 1 speed
Dispensing 100.0 uL into <Well B12>
Dropping tip <Well A1>
```

One to Many

You can transfer from a single source to multiple destinations, and the other way around (many sources to one destination).

```
pipette.transfer(100, plate.wells('A1'), plate.rows('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <WellSeries: <Well A2><Well B2><Well C2><Well D2>
↪<Well E2><Well F2><Well G2><Well H2>>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well A2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well C2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well D2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well E2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well F2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well G2>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well H2>
Dropping tip <Well A1>
```

Few to Many

What happens if, for example, you tell your pipette to transfer from 4 source wells to 2 destination wells? The transfer command will attempt to divide the wells evenly, or raise an error if the number of wells aren't divisible.

```
pipette.transfer(
    100,
    plate.wells('A1', 'A2', 'A3', 'A4'),
    plate.wells('B1', 'B2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <WellSeries: <Well A1><Well A2><Well A3><Well A4>> to
↪<WellSeries: <Well B1><Well B2>>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B1>
Aspirating 100.0 uL from <Well A2> at 1 speed
Dispensing 100.0 uL into <Well B1>
Aspirating 100.0 uL from <Well A3> at 1 speed
Dispensing 100.0 uL into <Well B2>
Aspirating 100.0 uL from <Well A4> at 1 speed
Dispensing 100.0 uL into <Well B2>
Dropping tip <Well A1>
```

List of Volumes

Instead of applying a single volume amount to all source/destination wells, you can instead pass a list of volumes.

```
pipette.transfer(
    [20, 40, 60],
    plate.wells('A1'),
    plate.wells('B1', 'B2', 'B3'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring [20, 40, 60] from <Well A1> to <WellSeries: <Well B1><Well B2><Well B3>>
Picking up tip <Well A1>
Aspirating 20.0 uL from <Well A1> at 1 speed
Dispensing 20.0 uL into <Well B1>
Aspirating 40.0 uL from <Well A1> at 1 speed
Dispensing 40.0 uL into <Well B2>
Aspirating 60.0 uL from <Well A1> at 1 speed
Dispensing 60.0 uL into <Well B3>
Dropping tip <Well A1>
```

Volume Gradient

Create a linear gradient between a start and ending volume (uL). The start and ending volumes must be the first and second elements of a tuple.

```
pipette.transfer(
    (100, 30),
```

```
    plate.wells('A1'),
    plate.rows('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring (100, 30) from <Well A1> to <WellSeries: <Well A2><Well B2><Well C2>
↪<Well D2><Well E2><Well F2><Well G2><Well H2>>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well A2>
Aspirating 90.0 uL from <Well A1> at 1 speed
Dispensing 90.0 uL into <Well B2>
Aspirating 80.0 uL from <Well A1> at 1 speed
Dispensing 80.0 uL into <Well C2>
Aspirating 70.0 uL from <Well A1> at 1 speed
Dispensing 70.0 uL into <Well D2>
Aspirating 60.0 uL from <Well A1> at 1 speed
Dispensing 60.0 uL into <Well E2>
Aspirating 50.0 uL from <Well A1> at 1 speed
Dispensing 50.0 uL into <Well F2>
Aspirating 40.0 uL from <Well A1> at 1 speed
Dispensing 40.0 uL into <Well G2>
Aspirating 30.0 uL from <Well A1> at 1 speed
Dispensing 30.0 uL into <Well H2>
Dropping tip <Well A1>
```

Distribute and Consolidate

Save time and tips with the `distribute()` and `consolidate()` commands. These are nearly identical to `transfer()`, except that they will combine multiple transfer's into a single tip.

Consolidate

Volumes going to the same destination well are combined within the same tip, so that multiple aspirates can be combined to a single dispense.

```
pipette.consolidate(30, plate.rows('2'), plate.wells('A1'))

for c in robot.commands():
    print(c)
```

will print out...

```
Consolidating 30 from <WellSeries: <Well A2><Well B2><Well C2><Well D2><Well E2><Well␣
↪F2><Well G2><Well H2>> to <Well A1>
Transferring 30 from <WellSeries: <Well A2><Well B2><Well C2><Well D2><Well E2><Well␣
↪F2><Well G2><Well H2>> to <Well A1>
Picking up tip <Well A1>
Aspirating 30.0 uL from <Well A2> at 1 speed
Aspirating 30.0 uL from <Well B2> at 1 speed
```

```
Aspirating 30.0 uL from <Well C2> at 1 speed
Aspirating 30.0 uL from <Well D2> at 1 speed
Aspirating 30.0 uL from <Well E2> at 1 speed
Aspirating 30.0 uL from <Well F2> at 1 speed
Dispensing 180.0 uL into <Well A1>
Aspirating 30.0 uL from <Well G2> at 1 speed
Aspirating 30.0 uL from <Well H2> at 1 speed
Dispensing 60.0 uL into <Well A1>
Dropping tip <Well A1>
```

If there are multiple destination wells, the pipette will never combine their volumes into the same tip.

```
pipette.consolidate(30, plate.rows('2'), plate.wells('A1', 'A2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Consolidating 30 from <WellSeries: <Well A2><Well B2><Well C2><Well D2><Well E2><Well␣
↪F2><Well G2><Well H2>> to <WellSeries: <Well A1><Well A2>>
Transferring 30 from <WellSeries: <Well A2><Well B2><Well C2><Well D2><Well E2><Well␣
↪F2><Well G2><Well H2>> to <WellSeries: <Well A1><Well A2>>
Picking up tip <Well A1>
Aspirating 30.0 uL from <Well A2> at 1 speed
Aspirating 30.0 uL from <Well B2> at 1 speed
Aspirating 30.0 uL from <Well C2> at 1 speed
Aspirating 30.0 uL from <Well D2> at 1 speed
Dispensing 120.0 uL into <Well A1>
Aspirating 30.0 uL from <Well E2> at 1 speed
Aspirating 30.0 uL from <Well F2> at 1 speed
Aspirating 30.0 uL from <Well G2> at 1 speed
Aspirating 30.0 uL from <Well H2> at 1 speed
Dispensing 120.0 uL into <Well A2>
Dropping tip <Well A1>
```

Distribute

Volumes from the same source well are combined within the same tip, so that one aspirate can provide for multiple dispenses.

```
pipette.distribute(55, plate.wells('A1'), plate.rows('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Distributing 55 from <Well A1> to <WellSeries: <Well A2><Well B2><Well C2><Well D2>
↪<Well E2><Well F2><Well G2><Well H2>>
Transferring 55 from <Well A1> to <WellSeries: <Well A2><Well B2><Well C2><Well D2>
↪<Well E2><Well F2><Well G2><Well H2>>
Picking up tip <Well A1>
Aspirating 165.0 uL from <Well A1> at 1 speed
Dispensing 55.0 uL into <Well A2>
Dispensing 55.0 uL into <Well B2>
```

```
Dispensing 55.0 uL into <Well C2>
Aspirating 165.0 uL from <Well A1> at 1 speed
Dispensing 55.0 uL into <Well D2>
Dispensing 55.0 uL into <Well E2>
Dispensing 55.0 uL into <Well F2>
Aspirating 110.0 uL from <Well A1> at 1 speed
Dispensing 55.0 uL into <Well G2>
Dispensing 55.0 uL into <Well H2>
Dropping tip <Well A1>
```

If there are multiple source wells, the pipette will never combine their volumes into the same tip.

```
pipette.distribute(30, plate.wells('A1', 'A2'), plate.rows('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Distributing 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
Transferring 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
Picking up tip <Well A1>
Aspirating 120.0 uL from <Well A1> at 1 speed
Dispensing 30.0 uL into <Well A2>
Dispensing 30.0 uL into <Well B2>
Dispensing 30.0 uL into <Well C2>
Dispensing 30.0 uL into <Well D2>
Aspirating 120.0 uL from <Well A2> at 1 speed
Dispensing 30.0 uL into <Well E2>
Dispensing 30.0 uL into <Well F2>
Dispensing 30.0 uL into <Well G2>
Dispensing 30.0 uL into <Well H2>
Dropping tip <Well A1>
```

Disposal Volume

When dispensing multiple times from the same tip, it is recommended to aspirate an extra amount of liquid to be disposed of after distributing. This added `disposal_vol` can be set as an optional argument.

```
pipette.distribute(
    30,
    plate.wells('A1', 'A2'),
    plate.rows('2'),
    disposal_vol=10)    # include extra liquid to make dispenses more accurate

for c in robot.commands():
    print(c)
```

will print out...

```
Distributing 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
Transferring 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
```

```
Picking up tip <Well A1>
Aspirating 130.0 uL from <Well A1> at 1 speed
Dispensing 30.0 uL into <Well A2>
Dispensing 30.0 uL into <Well B2>
Dispensing 30.0 uL into <Well C2>
Dispensing 30.0 uL into <Well D2>
Blowing out at <Well A1>
Aspirating 130.0 uL from <Well A2> at 1 speed
Dispensing 30.0 uL into <Well E2>
Dispensing 30.0 uL into <Well F2>
Dispensing 30.0 uL into <Well G2>
Dispensing 30.0 uL into <Well H2>
Blowing out at <Well A1>
Dropping tip <Well A1>
```

Note: If you do not specify a `disposal_vol`, the pipette will by default use a `disposal_vol` equal to it's `min_volume`. This tutorial has not given the pipette any `min_volume`, so below is an example of allowing the pipette's `min_volume` to be used as a default for `disposal_vol`.

```
pipette.min_volume = 20  # `min_volume` is used as default to `disposal_vol`

pipette.distribute(
    30,
    plate.wells('A1', 'A2'),
    plate.rows('2'))

for c in robot.commands():
    print(c)
```

will print out...

```
Distributing 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
Transferring 30 from <WellSeries: <Well A1><Well A2>> to <WellSeries: <Well A2><Well␣
↪B2><Well C2><Well D2><Well E2><Well F2><Well G2><Well H2>>
Picking up tip <Well A1>
Aspirating 140.0 uL from <Well A1> at 1 speed
Dispensing 30.0 uL into <Well A2>
Dispensing 30.0 uL into <Well B2>
Dispensing 30.0 uL into <Well C2>
Dispensing 30.0 uL into <Well D2>
Blowing out at <Well A1>
Aspirating 140.0 uL from <Well A2> at 1 speed
Dispensing 30.0 uL into <Well E2>
Dispensing 30.0 uL into <Well F2>
Dispensing 30.0 uL into <Well G2>
Dispensing 30.0 uL into <Well H2>
Blowing out at <Well A1>
Dropping tip <Well A1>
```

Transfer Options

There are other options for customizing your transfer command:

Always Get a New Tip

Transfer commands will by default use the same one tip for each well, then finally drop it in the trash once finished.

The pipette can optionally get a new tip at the beginning of each aspirate, to help avoid cross contamination.

```
pipette.transfer(
    100,
    plate.wells('A1', 'A2', 'A3'),
    plate.wells('B1', 'B2', 'B3'),
    new_tip='always')    # always pick up a new tip

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <WellSeries: <Well A1><Well A2><Well A3>> to <WellSeries: <Well␣
↪B1><Well B2><Well B3>>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B1>
Dropping tip <Well A1>
Picking up tip <Well B1>
Aspirating 100.0 uL from <Well A2> at 1 speed
Dispensing 100.0 uL into <Well B2>
Dropping tip <Well A1>
Picking up tip <Well C1>
Aspirating 100.0 uL from <Well A3> at 1 speed
Dispensing 100.0 uL into <Well B3>
Dropping tip <Well A1>
```

Never Get a New Tip

For scenarios where you instead are calling `pick_up_tip()` and `drop_tip()` elsewhere in your protocol, the transfer command can ignore picking up or dropping tips.

```
pipette.transfer(
    100,
    plate.wells('A1', 'A2', 'A3'),
    plate.wells('B1', 'B2', 'B3'),
    new_tip='never')    # never pick up or drop a tip

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <WellSeries: <Well A1><Well A2><Well A3>> to <WellSeries: <Well␣
↪B1><Well B2><Well B3>>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B1>
Aspirating 100.0 uL from <Well A2> at 1 speed
Dispensing 100.0 uL into <Well B2>
Aspirating 100.0 uL from <Well A3> at 1 speed
Dispensing 100.0 uL into <Well B3>
```

Trash or Return Tip

By default, the transfer command will drop the pipette's tips in the trash container. However, if you wish to instead return the tip to it's tip rack, you can set `trash=False`.

```
pipette.transfer(
    100,
    plate.wells('A1'),
    plate.wells('B1'),
    trash=False)       # do not trash tip

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <Well B1>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well B1>
Returning tip
Dropping tip <Well A1>
```

Touch Tip

A touch-tip can be performed after every aspirate and dispense by setting `touch_tip=True`.

```
pipette.transfer(
    100,
    plate.wells('A1'),
    plate.wells('A2'),
    touch_tip=True)     # touch tip to each well's edge

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <Well A2>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Touching tip
Dispensing 100.0 uL into <Well A2>
Touching tip
Dropping tip <Well A1>
```

Blow Out

A blow-out can be performed after every dispense that leaves the tip empty by setting `blow_out=True`.

```
pipette.transfer(
    100,
    plate.wells('A1'),
    plate.wells('A2'),
```

```
    blow_out=True)      # blow out droplets when tip is empty

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <Well A2>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well A2>
Blowing out
Dropping tip <Well A1>
```

Mix Before/After

A mix can be performed before every aspirate by setting `mix_before=`. The value of `mix_before=` must be a tuple, the 1st value is the number of repetitions, the 2nd value is the amount of liquid to mix.

```
pipette.transfer(
    100,
    plate.wells('A1'),
    plate.wells('A2'),
    mix_before=(2, 50), # mix 2 times with 50uL before aspirating
    mix_after=(3, 75))  # mix 3 times with 75uL after dispensing

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <Well A2>
Picking up tip <Well A1>
Mixing 2 times with a volume of 50ul
Aspirating 50 uL from <Well A1> at 1.0 speed
Dispensing 50 uL into None
Aspirating 50 uL from None at 1.0 speed
Dispensing 50 uL into None
Aspirating 100.0 uL from <Well A1> at 1 speed
Dispensing 100.0 uL into <Well A2>
Mixing 3 times with a volume of 75ul
Aspirating 75 uL from <Well A2> at 1.0 speed
Dispensing 75 uL into None
Aspirating 75 uL from None at 1.0 speed
Dispensing 75 uL into None
Aspirating 75 uL from None at 1.0 speed
Dispensing 75 uL into None
Dropping tip <Well A1>
```

Air Gap

An air gap can be performed after every aspirate by setting `air_gap=int`, where the value is the volume of air in microliters to aspirate after aspirating the liquid.

```
pipette.transfer(
    100,
    plate.wells('A1'),
    plate.wells('A2'),
    air_gap=20)          # add 20uL of air after each aspirate

for c in robot.commands():
    print(c)
```

will print out...

```
Transferring 100 from <Well A1> to <Well A2>
Picking up tip <Well A1>
Aspirating 100.0 uL from <Well A1> at 1 speed
Air gap
Aspirating 20 uL from None at 1.0 speed
Dispensing 20 uL into <Well A2>
Dispensing 100.0 uL into <Well A2>
Dropping tip <Well A1>
```

```
from opentrons import robot
robot.reset()
```

Advanced Control

Note: The below features are designed for advanced users who wish to use the Opentrons API in their own Python environment (ie Jupyter). This page is not relevant for users only using the Opentrons App, because the features described below will not be accessible.

The robot module can be thought of as the parent for all aspects of the Opentrons API. All containers, instruments, and protocol commands are added to and controlled by robot.

```
'''
Examples in this section require the following
'''
from opentrons import robot, labware, instruments

plate = labware.load('96-flat', 'B1', 'my-plate')
tiprack = labware.load('tiprack-200ul', 'A1', 'my-rack')

pipette = instruments.P300_Single(mount='left', tip_racks=[tiprack])
```

User-Specified Pause

This will pause your protocol at a specific step. You can resume by pressing ‘resume’ in your OT App.

```
robot.pause()
```

Head Speed

The speed of the robot's motors can be set using `robot.head_speed()`. The units are all millimeters-per-second (mm/sec). The `x`, `y`, `z`, `a`, `b`, `c` parameters set the maximum speed of the corresponding axis on Smoothie.

'x': lateral motion, 'y': front to back motion, 'z': vertical motion of the left mount, 'a': vertical motion of the right mount, 'b': plunger motor for the left pipette, 'c': plunger motor for the right pipette.

The `combined_speed` parameter sets the speed across all axes to either the specified value or the axis max, whichever is lower. Defaults are specified by `DEFAULT_MAX_SPEEDS` in robot_configs.py[20].

```
max_speed_per_axis = {
    'x': 600, 'y': 400, 'z': 125, 'a': 125, 'b': 50, 'c': 50}
robot.head_speed(
    combined_speed=max(max_speed_per_axis.values()),
    **max_speed_per_axis)
```

Homing

You can *home* the robot by calling `home()`. You can also specify axes. The robot will home immdediately when this call is made.

```
robot.home()            # home the robot on all axis
robot.home('z')         # home the Z axis only
```

Commands

When commands are called on a pipette, they are recorded on the `robot` in the order they are called. You can see all past executed commands by calling `robot.commands()`, which returns a Python list[21].

```
pipette.pick_up_tip(tiprack.wells('A1'))
pipette.drop_tip(tiprack.wells('A1'))

for c in robot.commands():
    print(c)
```

will print out...

```
Picking up tip <Well A1>
Dropping tip <Well A1>
```

Clear Commands

We can erase the robot command history by calling `robot.clear_commands()`. Any previously created instruments and containers will still be inside robot, but the commands history is erased.

```
robot.clear_commands()
pipette.pick_up_tip(tiprack['A1'])
print('There is', len(robot.commands()), 'command')
```

[20] https://github.com/Opentrons/opentrons/blob/edge/api/src/opentrons/config/robot_configs.py
[21] https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists

```
robot.clear_commands()
print('There are now', len(robot.commands()), 'commands')
```

will print out...

```
There is 1 command
There are now 0 commands
```

Comment

You can add a custom message to the list of command descriptions you see when running `robot.commands()`. This command is `robot.comment()`, and it allows you to print out any information you want at the point in your protocol

```
robot.clear_commands()

pipette.pick_up_tip(tiprack['A1'])
robot.comment("Hello, just picked up tip A1")

pipette.pick_up_tip(tiprack['A1'])
robot.comment("Goodbye, just dropped tip A1")

for c in robot.commands():
    print(c)
```

will print out...

```
Picking up tip <Well A1>
Hello, just picked up tip A1
Picking up tip <Well A1>
Goodbye, just dropped tip A1
```

Get Containers

When containers are loaded, they are automatically added to the `robot`. You can see all currently held containers by calling `robot.get_containers()`, which returns a Python list[22].

```
for container in robot.get_containers():
    print(container.get_name(), container.get_type())
```

will print out...

```
my-rack tiprack-200ul
my-plate 96-flat
```

Reset

Calling `robot.reset()` will remove everything from the robot. Any previously added containers, pipettes, or commands will be erased.

[22] https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists

```
robot.reset()
print(robot.get_containers())
print(robot.commands())
```

will print out...

```
[]
[]
[]
```

Hardware Modules

This documentation and modules API is subject to change. Check here or on our github for updated information.

This code is only valid on software version 3.3.0 or later.

Loading your Module onto a deck

Just like labware, you will also need to load in your module in order to use it within a protocol. To do this, you call:

```
from opentrons import modules

module = modules.load('Module Name', slot)
```

Above, *Module Name* represents either *tempdeck* or *magdeck*.

To add a labware onto a given module, you will need to use the *share=True* call-out

```
from opentrons import labware

labware = labware.load('96-flat', slot, share=True)
```

Where slot is the same slot in which you loaded your module.

Detecting your Module on the robot

The Run App auto-detects and connects to modules that are plugged into the robot upon robot connection. If you plug in a module with the app open and connected to your robot already, you can simply navigate to the *Pipettes & Modules* in the Run App and hit the *refresh* icon.

If you are running a program outside of the app, you will need to initiate robot connection to the module. This can be done like the following:

```
from opentrons import modules, robot

robot.connect()
robot.discover_modules()

module = modules.load('Module Name', slot)
... etc
```

Checking the status of your Module

Both modules have the ability to check what state they are currently in. To do this run the following:

```
from opentrons import modules

module = modules.load('Module Name', slot)
status = module.status
```

For the temperature module this will return a string stating whether it's *heating*, *cooling*, *holding at target* or *idle*. For the magnetic module this will return a string stating whether it's *engaged* or *disengaged*.

Temperature Module

Our temperature module acts as both a cooling and heating device. The range of temperatures this module can reach goes from 4 to 95 degrees celsius with a resolution of 1 degree celcius.

The temperature module has the following methods that can be accessed during a protocol.

Set Temperature

To set the temperature module to a given temperature in degrees celsius do the following:

```
from opentrons import modules, labware

module = modules.load('tempdeck', slot)
plate = labware.load('96-flat', slot, share=True)

module.set_temperature(4)
```

This will set your Temperature module to 4 degrees celsius.

Wait Until Setpoint Reached

This function will pause your protocol until your target temperature is reached.

```
from opentrons import modules, labware

module = modules.load('tempdeck', slot)
plate = labware.load('96-flat', slot, share=True)

module.set_temperature(4)
module.wait_for_temp()
```

Before using *wait_for_temp()* you must set a target temperature with *set_temperature()*. Once the target temperature is set, when you want the protocol to wait until the module reaches the target you can call *wait_for_temp()*.

If no target temperature is set via *set_temperature()*, the protocol will be stuck in an indefinite loop.

Read the Current Temperature

You can read the current real-time temperature of the module by the following:

```
from opentrons import modules, labware

module = modules.load('tempdeck', slot)
plate = labware.load('96-flat', slot, share=True)

temperature = module.temperature
```

This will return a float of the temperature in celsius.

Read the Target Temperature

We can read the target temperature of the module by the following:

```
from opentrons import modules, labware

module = modules.load('tempdeck', slot)
plate = labware.load('96-flat', slot, share=True)

temperature = module.target
```

This will return a float of the temperature that the module is trying to reach.

Deactivate

This function will stop heating or cooling and will turn off the fan on the module. You would still be able to call *set_temperature()* function to initiate a heating or cooling phase again.

```
from opentrons import modules, labware

module = modules.load('tempdeck', slot)
plate = labware.load('96-flat', slot, share=True)

module.set_temperature(4)
module.wait_for_temp()

## OTHER PROTOCOL ACTIONS

module.deactivate()
```

** Note** You can also deactivate your temperature module through our Run App by clicking on the *Pipettes & Modules* tab. Your temperature module will automatically deactivate if another protocol is uploaded to the app. Your temperature module will not deactivate automatically upon protocol end, cancel or re-setting a protocol.

Magnetic Module

The magnetic module has two actions:

- engage: The magnetic stage rises to a default height unless an *offset* or a custom *height* is specified
- disengage: The magnetic stage moves down to its home position

The magnetic module api is currently fully compatible with the BioRad Hardshell 96-PCR (.2ml) well plates. The magnets will default to an engaged height of about 4.3 mm from the bottom of the well (or 18mm from magdeck home position). This is roughly 30% of the well depth. This engaged height has been tested for an elution volume of 40ul.

You can also specify a custom engage height for the magnets so you can use a different labware with the magdeck. In the future, we will have adapters to support tuberacks as well as deep well plates.

Engage

```
from opentrons import modules, labware

module = modules.load('magdeck', slot)
plate = labware.load('biorad-hardshell-96-PCR', slot, share=True)

module.engage()
```

If you deem that the default engage height is not ideal for your applications, you can include an offset in mm for the magnet to move to. The engage function will take in a value (positive or negative) to offset the magnets from the **default** position.

To move the magnets higher than the default position you would specify a positive mm offset such as: `module.engage(offset=4)`

To move the magnets lower than the default position you would input a negative mm value such as: `module.engage(offset=-4)`

You can also use a custom height parameter with engage():

```
from opentrons import modules, labware

module = modules.load('magdeck', slot)
plate = labware.load('96-deep-well', slot, share=True)

module.engage(height=12)
```

The height should be specified in mm from the magdeck home position (i.e. the position of magnets when power-cycled or disengaged)

** Note ** *engage()* and *engage(offset=y)* can only be used for labware that have default heights defined in the api. If your labware doesn't yet have a default height definition and your protocol uses either of those methods then you will get an error. Simply use the height parameter to provide a custom height for you labware in such a case.

Disengage

```
from opentrons import modules, labware

module = modules.load('magdeck', slot)
plate = labware.load('biorad-hardshell-96-PCR', slot, share=True)

module.engage()
## OTHER PROTOCOL ACTIONS
module.disengage()
```

The magnetic modules will disengage on power cycle of the device. It will not auto-disengage otherwise unless you specify in your protocol.

Examples

All examples on this page assume the following labware and pipette:

```
from opentrons import robot, labware, instruments

plate = labware.load('96-flat', '1')
trough = labware.load('trough-12row', '2')

tiprack_1 = labware.load('tiprack-200ul', '3')
tiprack_2 = labware.load('tiprack-200ul', '4')

p300 = instruments.P300_Single(
    mount='left',
    tip_racks=[tiprack_2])
```

Basic Transfer

Moving 100uL from one well to another:

```
p300.transfer(100, plate.wells('A1'), plate.wells('B1'))
```

If you prefer to not use the `.transfer()` command, the following pipette commands will create the some results:

```
p300.pick_up_tip()
p300.aspirate(100, plate.wells('A1'))
p300.dispense(100, plate.wells('A1'))
p300.return_tip()
```

Loops

Loops in Python allows your protocol to perform many actions, or act upon many wells, all within just a few lines. The below example loops through the numbers `0` to `11`, and uses that loop's current value to transfer from all wells in a trough to each row of a plate:

```
# distribute 20uL from trough:A1 -> plate:row:1
# distribute 20uL from trough:A2 -> plate:row:2
# etc...

# ranges() starts at 0 and stops at 12, creating a range of 0-11
for i in range(12):
  p300.distribute(200, trough.wells(i), plate.rows(i))
```

Multiple Air Gaps

The Opentrons liquid handler can do some things that a human cannot do with a pipette, like accurately alternate between aspirating and creating air gaps within the same tip. The below example will aspirate from five wells in the trough, while creating a air gap between each sample.

```
p300.pick_up_tip()

for well in trough.wells():
  p300.aspirate(35, well).air_gap(10)

p300.dispense(plate.wells('A1'))

p300.return_tip()
```

Dilution

This example first spreads a dilutent to all wells of a plate. It then dilutes 8 samples from the trough across the 8 columns of the plate.

```
p300.distribute(50, trough.wells('A12'), plate.wells())  # dilutent

# loop through each row
for i in range(8):

  # save the source well and destination column to variables
  source = trough.wells(i)
  row = plate.rows(i)

  # transfer 30uL of source to first well in column
  p300.transfer(30, source, column.wells('1'))

  # dilute the sample down the column
  p300.transfer(
    30, row.wells('1', to='11'), row.wells('2', to='12'),
    mix_after=(3, 25))
```

Plate Mapping

Deposit various volumes of liquids into the same plate of wells, and automatically refill the tip volume when it runs out.

```
# these uL values were created randomly for this example
water_volumes = [
  1,  2,  3,  4,  5,  6,  7,  8,
  9,  10, 11, 12, 13, 14, 15, 16,
  17, 18, 19, 20, 21, 22, 23, 24,
  25, 26, 27, 28, 29, 30, 31, 32,
  33, 34, 35, 36, 37, 38, 39, 40,
  41, 42, 43, 44, 45, 46, 47, 48,
  49, 50, 51, 52, 53, 54, 55, 56,
  57, 58, 59, 60, 61, 62, 63, 64,
  65, 66, 67, 68, 69, 70, 71, 72,
  73, 74, 75, 76, 77, 78, 79, 80,
  81, 82, 83, 84, 85, 86, 87, 88,
  89, 90, 91, 92, 93, 94, 95, 96
]
```

```
p300.distribute(water_volumes, trough.wells('A12'), plate)
```

The final volumes can also be read from a CSV, and opened by your protocol.

```
'''
  This example uses a CSV file saved on the same computer, formatted as follows,
  where the columns in the file represent the 12 columns of the plate,
  and the rows in the file represent the 8 rows of the plate,
  and the values represent the uL that must end up at that location

  1,  2,  3,  4,  5,  6,  7,  8, 9,  10, 11, 12,
  13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
  25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
  37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
  49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
  61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
  73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
  85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96
'''

# open file with absolute path (will be different depending on operating system)
# file paths on Windows look more like 'C:\\path\\to\\your\\csv_file.csv'
with open('/path/to/your/csv_file.csv') as my_file:

    # save all volumes from CSV file into a list
    volumes = []

    # loop through each line (the plate's columns)
    for l in my_file.read().splitlines():
        # loop through each comma-separated value (the plate's rows)
        for v in l.split(','):
            volumes.append(float(v))  # save the volume

    # distribute those volumes to the plate
    p300.distribute(volumes, trough.wells('A1'), plate.wells())
```

Precision Pipetting

This example shows how to deposit liquid around the edge of a well.

```
p300.pick_up_tip()
p300.aspirate(200, trough.wells('A1'))
# rotate around the edge of the well, dropping 20ul at a time
theta = 0.0
while p300.current_volume > 0:
    # we can move around a circle with radius (r) and theta (degrees)
    well_edge = plate.wells('B1').from_center(r=1.0, theta=theta, h=0.9)

    # combine a Well with a Vector in a tuple
    destination = (plate.wells('B1'), well_edge)
    p300.move_to(destination, strategy='direct')  # move straight there
    p300.dispense(20)
```

```
    theta += 0.314

p300.drop_tip()
```

API Reference

If you are reading this, you are probably looking for an in-depth explanation of API classes and methods to fully master your protocol development skills.

Robot

All protocols are set up, simulated and executed using a Robot class.

class `opentrons.legacy_api.robot.`**Robot** (*config=None*, *broker=None*)

This class is the main interface to the robot.

It should never be instantiated directly; instead, the global instance may be accessed at `opentrons.robot`.

Through this class you can can:

- define your `opentrons.Deck`
- `connect()` to Opentrons physical robot
- `home()` axis, move head (`move_to()`)
- `pause()` and `resume()` the protocol run
- set the `head_speed()` of the robot

Each Opentrons protocol is a Python script. When evaluated the script creates an execution plan which is stored as a list of commands in Robot's command queue.

Here are the typical steps of writing the protocol:

- Using a Python script and the Opentrons API load your containers and define instruments (see `Pipette`).
- Call `reset()` to reset the robot's state and clear commands.
- Write your instructions which will get converted into an execution plan.
- Review the list of commands generated by a protocol `commands()`.
- `connect()` to the robot and call `run()` it on a real robot.

See `Pipette` for the list of supported instructions.

add_instrument (*mount*, *instrument*)

Adds instrument to a robot.

Parameters

- **mount** (*str*) – Specifies which axis the instruments is attached to. Valid options are "left" or "right".
- **instrument** (*Instrument*) – An instance of a `Pipette` to attached to the axis.

Notes

A canonical way to add to add a Pipette to a robot is:

```
from opentrons import instruments
m300 = instruments.P300_Multi(mount='left')
```

This will create a pipette and call `add_instrument()` to attach the instrument.

connect(*port=None*, *options=None*)
Connects the robot to a serial port.

Parameters

- **port** (*str*) – OS-specific port name or `'Virtual Smoothie'`
- **options** (*dict*) – if `port` is set to `'Virtual Smoothie'`, provide the list of options to be passed to `get_virtual_device()`

Returns

Return type `True` for success, `False` for failure.

Note: If you wish to connect to the robot without using the OT App, you will need to use this function.

Examples

```
>>> from opentrons import robot
>>> robot.connect()
```

disconnect()
Disconnects from the robot.

get_warnings()
Get current runtime warnings.

Returns

- Runtime warnings accumulated since the last `run()`
- or *simulate()* (page 73).

head_speed(*combined_speed=None*, *x=None*, *y=None*, *z=None*, *a=None*, *b=None*, *c=None*)
Set the speeds (mm/sec) of the robot

Parameters

- **combined_speed** (*number specifying a combined-axes speed*) –
- **<axis>** (*key/value pair, specifying the maximum speed of that axis*) –

Examples

```
>>> from opentrons import robot
>>> robot.reset()
>>> robot.head_speed(combined_speed=400)
```

```
#  sets the head speed to 400 mm/sec or the axis max per axis
>>> robot.head_speed(x=400, y=200)
# sets max speeds of X and Y
```

home (**args*, ***kwargs*)
Home robot's head and plunger motors.

move_to (*location*, *instrument*, *strategy='arc'*, ***kwargs*)
Move an instrument to a coordinate, container or a coordinate within a container.

Parameters

- **location** (*one of the following:*) – 1. `Placeable` (i.e. Container, Deck, Slot, Well) — will move to the origin of a container. 2. `Vector` move to the given coordinate in Deck coordinate system. 3. (`Placeable`, `Vector`) move to a given coordinate within object's coordinate system.
- **instrument** – Instrument to move relative to. If `None`, move relative to the center of a gantry.
- **strategy** (*{'arc', 'direct'}*) – `arc` : move to the point using arc trajectory avoiding obstacles.

  `direct` : move to the point in a straight line.

pause (*msg=None*)
Pauses execution of the protocol. Use `resume()` to resume

reset ()

Resets the state of the robot and clears:

- Deck
- Instruments
- Command queue
- Runtime warnings

Examples

```
>>> from opentrons import robot
>>> robot.reset()
```

resume ()
Resume execution of the protocol after `pause()`

stop ()
Stops execution of the protocol. (alias for *halt*)

Pipette

class opentrons.legacy_api.instruments.**Pipette**(*robot*, *model_offset=(0, 0, 0)*, *mount=None*, *axis=None*, *mount_obj=None*, *name=None*, *ul_per_mm=None*, *channels=1*, *min_volume=0*, *max_volume=None*, *trash_container=''*, *tip_racks=[]*, *aspirate_speed=5*, *dispense_speed=10*, *aspirate_flow_rate=None*, *dispense_flow_rate=None*, *plunger_current=0.5*, *drop_tip_current=0.5*, *drop_tip_speed=5*, *plunger_positions={'top': 18.5, 'bottom': 2, 'blow_out': 0, 'drop_tip': -3.5}*, *pick_up_current=0.1*, *pick_up_distance=10*, *pick_up_increment=1*, *pick_up_presses=3*, *pick_up_speed=30*, *quirks=[]*, *fallback_tip_length=51.7*)

DIRECT USE OF THIS CLASS IS DEPRECATED – this class should not be used directly. Its parameters, defaults, methods, and behaviors are subject to change without a major version release. Use the model-specific constructors available through `from opentrons import instruments`.

All model-specific instrument constructors are inheritors of this class. With any of those instances you can can:

- Handle liquids with *aspirate()* (page 66), *dispense()* (page 67), *mix()* (page 69), and *blow_out()* (page 66)
- Handle tips with *pick_up_tip()* (page 70), *drop_tip()* (page 68), and *return_tip()* (page 71)
- Calibrate this pipette's plunger positions
- Calibrate the position of each `Container` on deck

Here are the typical steps of using the Pipette:

- Instantiate a pipette with a maximum volume (uL)

and a mount (*left* or *right*) * Design your protocol through the pipette's liquid-handling commands

Methods in this class include assertions where needed to ensure that any action that requires a tip must be preceeded by *pick_up_tip*. For example: *mix*, *transfer*, *aspirate*, *blow_out*, and *drop_tip*.

Parameters

- **mount** (*str*) – The mount of the pipette's actuator on the Opentrons robot ('left' or 'right')
- **trash_container** (*Container*) – Sets the default location *drop_tip()* (page 68) will put tips (Default: *fixed-trash*)
- **tip_racks** (*list*) – A list of Containers for this Pipette to track tips when calling *pick_up_tip()* (page 70) (Default: [])
- **aspirate_flow_rate** (*int*) – The speed (in ul/sec) the plunger will move while aspirating (Default: See Model Type)
- **dispense_flow_rate** (*int*) – The speed (in ul/sec) the plunger will move while dispensing (Default: See Model Type)

Returns

Return type A new instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> tip_rack_300ul = labware.load(
...     'GEB-tiprack-300ul', '1')
>>> p300 = instruments.P300_Single(mount='left',
...     tip_racks=[tip_rack_300ul])
```

aspirate (*volume=None*, *location=None*, *rate=1.0*)

Aspirate a volume of liquid (in microliters/uL) using this pipette from the specified location

Notes

If only a volume is passed, the pipette will aspirate from it's current position. If only a location is passed, *aspirate* will default to it's *max_volume*.

The location may be a Well, or a specific position in relation to a Well, such as *Well.top()*. If a Well is specified without calling a a position method (such as .top or .bottom), this method will default to the bottom of the well.

Parameters

- **volume** (*int or float*) – The number of microliters to aspirate (Default: self.max_volume)
- **location** (`Placeable` or tuple(`Placeable`, `Vector`)) – The `Placeable` (`Well`) to perform the aspirate. Can also be a tuple with first item `Placeable`, second item relative `Vector`
- **rate** (*float*) – Set plunger speed for this aspirate, where speed = rate * aspirate_speed (see `set_speed()`)

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', '2')
>>> p300 = instruments.P300_Single(mount='right')
>>> p300.pick_up_tip()
# aspirate 50uL from a Well
>>> p300.aspirate(50, plate[0])
# aspirate 50uL from the center of a well
>>> p300.aspirate(50, plate[1].bottom())
>>> # aspirate 20uL in place, twice as fast
>>> p300.aspirate(20, rate=2.0)
>>> # aspirate the pipette's remaining volume (80uL) from a Well
>>> p300.aspirate(plate[2])
```

blow_out (*location=None*)
Force any remaining liquid to dispense, by moving this pipette's plunger to the calibrated *blow_out* position

Notes

If no *location* is passed, the pipette will blow_out from it's current position.

Parameters **location** (`Placeable` or tuple(`Placeable`, `Vector`)) – The `Placeable` (`Well`) to perform the blow_out. Can also be a tuple with first item `Placeable`, second item relative `Vector`

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, robot
>>> robot.reset()
>>> p300 = instruments.P300_Single(mount='left')
>>> p300.aspirate(50).dispense().blow_out()
```

consolidate (*volume*, *source*, *dest*, **args*, ***kwargs*)
Consolidate will move a volume of liquid from a list of sources to a single target location. See `Transfer` for details and a full list of optional arguments.

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', 'A3')
>>> p300 = instruments.P300_Single(mount='left')
>>> p300.consolidate(50, plate.cols[0], plate[1])
```

delay (*seconds=0*, *minutes=0*)

Parameters **seconds** (*float*) – The number of seconds to freeze in place.

dispense (*volume=None*, *location=None*, *rate=1.0*)
Dispense a volume of liquid (in microliters/uL) using this pipette

Notes

If only a volume is passed, the pipette will dispense from it's current position. If only a location is passed, *dispense* will default to it's *current_volume*

The location may be a Well, or a specific position in relation to a Well, such as *Well.top()*. If a Well is specified without calling a a position method (such as .top or .bottom), this method will default to the bottom of the well.

Parameters

- **volume** (*int or float*) – The number of microliters to dispense (Default: self.current_volume)
- **location** (`Placeable` or tuple(`Placeable`, `Vector`)) – The `Placeable` (`Well`) to perform the dispense. Can also be a tuple with first item `Placeable`, second item relative `Vector`
- **rate** (*float*) – Set plunger speed for this dispense, where speed = rate * dispense_speed (see `set_speed()`)

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', '3')
>>> p300 = instruments.P300_Single(mount='left')
# fill the pipette with liquid (200uL)
>>> p300.aspirate(plate[0])
# dispense 50uL to a Well
>>> p300.dispense(50, plate[0])
# dispense 50uL to the center of a well
>>> relative_vector = plate[1].center()
>>> p300.dispense(50, (plate[1], relative_vector))
# dispense 20uL in place, at half the speed
>>> p300.dispense(20, rate=0.5)
# dispense the pipette's remaining volume (80uL) to a Well
>>> p300.dispense(plate[2])
```

distribute(*volume*, *source*, *dest*, **args*, ***kwargs*)

Distribute will move a volume of liquid from a single of source to a list of target locations. See `Transfer` for details and a full list of optional arguments.

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', '3')
>>> p300 = instruments.P300_Single(mount='left')
>>> p300.distribute(50, plate[1], plate.cols[0])
```

drop_tip(*location=None*, *home_after=True*)

Drop the pipette's current tip

Notes

If no location is passed, the pipette defaults to its *trash_container* (see *Pipette* (page 65))

Parameters **location** (Placeable or tuple(Placeable, Vector)) – The Placeable (Well) to perform the drop_tip. Can also be a tuple with first item Placeable, second item relative Vector

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> tiprack = labware.load('tiprack-200ul', 'C2')
>>> trash = labware.load('point', 'A3')
>>> p300 = instruments.P300_Single(mount='left')
>>> p300.pick_up_tip(tiprack[0])
# drops the tip in the fixed trash
>>> p300.drop_tip()
>>> p300.pick_up_tip(tiprack[1])
# drops the tip back at its tip rack
>>> p300.drop_tip(tiprack[1])
```

home()

Home the pipette's plunger axis during a protocol run

Notes

Pipette.home() homes the *Robot*

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, robot
>>> robot.reset()
>>> p300 = instruments.P300_Single(mount='right')
>>> p300.home()
```

mix(*repetitions=1*, *volume=None*, *location=None*, *rate=1.0*)

Mix a volume of liquid (in microliters/uL) using this pipette

Notes

If no *location* is passed, the pipette will mix from it's current position. If no *volume* is passed, *mix* will default to it's *max_volume*

Parameters

- **repetitions** (*int*) – How many times the pipette should mix (Default: 1)
- **volume** (*int or float*) – The number of microliters to mix (Default: self.max_volume)

- **location** (Placeable or tuple(Placeable, Vector)) – The Placeable (Well) to perform the mix. Can also be a tuple with first item Placeable, second item relative Vector
- **rate** (*float*) – Set plunger speed for this mix, where speed = rate * (aspirate_speed or dispense_speed) (see set_speed())

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', '4')
>>> p300 = instruments.P300_Single(mount='left')
# mix 50uL in a Well, three times
>>> p300.mix(3, 50, plate[0])
# mix 3x with the pipette's max volume, from current position
>>> p300.mix(3)
```

move_to(*location*, *strategy=None*)

Move this *Pipette* (page 65) to a Placeable on the Deck

Notes

Until obstacle-avoidance algorithms are in place, *Robot* (page 62) and *Pipette* (page 65) *move_to()* (page 70) use either an "arc" or "direct"

Parameters

- **location** (Placeable or tuple(Placeable, Vector)) – The destination to arrive at
- **strategy** (*"arc" or "direct"*) – "arc" strategies (default) will pick the head up on Z axis, then over to the XY destination, then finally down to the Z destination. "direct" strategies will simply move in a straight line from the current position

Returns

Return type This instance of *Pipette* (page 65).

pick_up_tip(*location=None*, *presses=None*, *increment=None*)

Pick up a tip for the Pipette to run liquid-handling commands with

Notes

A tip can be manually set by passing a *location*. If no location is passed, the Pipette will pick up the next available tip in it's *tip_racks* list (see *Pipette* (page 65))

Parameters

- **location** (Placeable or tuple(Placeable, Vector)) – The Placeable (Well) to perform the pick_up_tip. Can also be a tuple with first item Placeable, second item relative Vector

- **presses** (*:any:int*) – The number of times to lower and then raise the pipette when picking up a tip, to ensure a good seal (0 [zero] will result in the pipette hovering over the tip but not picking it up–generally not desireable, but could be used for dry-run). Default: 3 presses
- **increment** (*:int*) – The additional distance to travel on each successive press (e.g.: if presses=3 and increment=1, then the first press will travel down into the tip by 3.5mm, the second by 4.5mm, and the third by 5.5mm. Default: 1mm

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> tiprack = labware.load('GEB-tiprack-300', '2')
>>> p300 = instruments.P300_Single(mount='left',
...     tip_racks=[tiprack])
>>> p300.pick_up_tip(tiprack[0])
>>> p300.return_tip()
# `pick_up_tip` will automatically go to tiprack[1]
>>> p300.pick_up_tip()
>>> p300.return_tip()
```

return_tip(*home_after=True*)

Drop the pipette's current tip to it's originating tip rack

Notes

This method requires one or more tip-rack `Container` to be in this Pipette's *tip_racks* list (see *Pipette* (page 65))

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> tiprack = labware.load('GEB-tiprack-300', '2')
>>> p300 = instruments.P300_Single(mount='left',
...     tip_racks=[tiprack, tiprack2])
>>> p300.pick_up_tip()
>>> p300.aspirate(50, plate[0])
>>> p300.dispense(plate[1])
>>> p300.return_tip()
```

set_flow_rate(*aspirate=None*, *dispense=None*)

Set the speed (uL/second) the *Pipette* (page 65) plunger will move during *aspirate()* (page 66) and *dispense()* (page 67). The speed is set using nominal max volumes for any given pipette model. :param aspirate: The speed in microliters-per-second, at which the plunger will

move while performing an aspirate

Parameters **dispense** (*int*) – The speed in microliters-per-second, at which the plunger will move while performing an dispense

touch_tip (*location=None*, *radius=1.0*, *v_offset=-1.0*, *speed=60.0*)

Touch the *Pipette* (page 65) tip to the sides of a well, with the intent of removing left-over droplets

Notes

If no *location* is passed, the pipette will touch_tip from it's current position.

Parameters

- **location** (Placeable or tuple(Placeable, Vector)) – The Placeable (Well) to perform the touch_tip. Can also be a tuple with first item Placeable, second item relative Vector
- **radius** (*float*) – Radius is a floating point describing the percentage of a well's radius. When radius=1.0, *touch_tip()* (page 72) will move to 100% of the wells radius. When radius=0.5, *touch_tip()* (page 72) will move to 50% of the wells radius. Default: 1.0 (100%)
- **speed** (*float*) – The speed for touch tip motion, in mm/s. Default: 60.0 mm/s, Max: 80.0 mm/s, Min: 20.0 mm/s
- **v_offset** (*float*) – The offset in mm from the top of the well to touch tip. Default: -1.0 mm

Returns

Return type This instance of *Pipette* (page 65).

Examples

```
>>> from opentrons import instruments, labware, robot
>>> robot.reset()
>>> plate = labware.load('96-flat', '8')
>>> p300 = instruments.P300_Single(mount='left')
>>> p300.aspirate(50, plate[0])
>>> p300.dispense(plate[1]).touch_tip()
```

transfer (*volume*, *source*, *dest*, ***kwargs*)

Transfer will move a volume of liquid from a source location(s) to a dest location(s). It is a higher-level command, incorporating other *Pipette* (page 65) commands, like *aspirate* (page 66) and *dispense* (page 67), designed to make protocol writing easier at the cost of specificity.

Parameters

- **volumes** (*number, list, or tuple*) – The amount of volume to remove from each *sources* Placeable and add to each *targets* Placeable. If *volumes* is a list, each volume will be used for the sources/targets at the matching index. If *volumes* is a tuple with two elements, like *(20, 100)*, then a list of volumes will be generated with a linear gradient between the two volumes in the tuple.
- **source** (*Placeable or list*) – Single Placeable or list of :any:‘Placeable‘s, from where liquid will be :any:‘aspirate‘ed from.

- **dest** (*Placeable or list*) – Single `Placeable` or list of :any:‘Placeable‘s, where liquid will be :any:‘dispense‘ed to.
- **new_tip** (*str*) – The number of clean tips this transfer command will use. If ‘never’, no tips will be picked up nor dropped. If ‘once’, a single tip will be used for all commands. If ‘always’, a new tip will be used for each transfer. Default is ‘once’.
- **trash** (*boolean*) – If *False* (default behavior) tips will be returned to their tip rack. If *True* and a trash container has been attached to this *Pipette*, then the tip will be sent to the trash container.
- **touch_tip** (*boolean*) – If *True*, a *touch_tip* (page 72) will occur following each *aspirate* (page 66) and *dispense* (page 67). If set to *False* (default), no *touch_tip* (page 72) will occur.
- **blow_out** (*boolean*) – If *True*, a *blow_out* (page 66) will occur following each *dispense* (page 67), but only if the pipette has no liquid left in it. If set to *False* (default), no *blow_out* (page 66) will occur.
- **mix_before** (*tuple*) – Specify the number of repetitions volume to mix, and a *mix* (page 69) will proceed each *aspirate* (page 66) during the transfer and dispense. The tuple’s values is interpreted as (repetitions, volume).
- **mix_after** (*tuple*) – Specify the number of repetitions volume to mix, and a *mix* (page 69) will following each *dispense* (page 67) during the transfer or consolidate. The tuple’s values is interpreted as (repetitions, volume).
- **carryover** (*boolean*) – If *True* (default), any *volumes* that exceed the maximum volume of this *Pipette* will be split into multiple smaller volumes.
- **repeat** (*boolean*) – (Only applicable to *distribute* (page 68) and *consolidate* (page 67))If *True* (default), sequential *aspirate* (page 66) volumes will be combined into one tip for the purpose of saving time. If *False*, all volumes will be transferred seperately.
- **gradient** (*lambda*) – Function for calculated the curve used for gradient volumes. When *volumes* is a tuple of length 2, it’s values are used to create a list of gradient volumes. The default curve for this gradient is linear (lambda x: x), however a method can be passed with the *gradient* keyword argument to create a custom curve.

Returns

Return type This instance of *Pipette* (page 65).

Examples

... >>> from opentrons import instruments, labware, robot # doctest: +SKIP >>> robot.reset() # doctest: +SKIP >>> plate = labware.load(‘96-flat’, ‘5’) # doctest: +SKIP >>> p300 = instruments.P300_Single(mount=’right’) # doctest: +SKIP >>> p300.transfer(50, plate[0], plate[1]) # doctest: +SKIP

Simulation

An easy entrypoint for simulating a protocol offline.

`opentrons.simulate.`**format_runlog**(*runlog: typing.List[typing.Mapping[str, typing.Any]]*) → str

Format a run log (return value of `simulate` ()) into a human-readable string

Parameters **runlog** – The output of a call to *simulate()* (page 74)

opentrons.simulate.**simulate**(*protocol_file*, *propagate_logs=False*, *log_level='warning'*) → typing.List[typing.Mapping[str, typing.Any]]

Simulate the protocol itself.

This is a one-stop function to simulate a protocol, whether python or json, no matter the api version, from external (i.e. not bound up in other internal server infrastructure) sources.

To simulate an opentrons protocol from other places, pass in a file like object as protocol_file; this function either returns (if the simulation has no problems) or raises an exception.

To call from the command line use either the autogenerated entrypoint `opentrons_simulate` (`opentrons_simulate.exe`, on windows) or `python -m opentrons.simulate`.

The return value is the run log, a list of dicts that represent the commands executed by the robot. Each dict has the following keys:

- **`level`: The depth at which this command is nested - if this an** aspirate inside a mix inside a transfer, for instance, it would be 3.
- **`payload`: The command, its arguments, and how to format its text.** For more specific details see `opentrons.commands`. To format a message from a payload do `payload['text'].format(**payload)`.
- **`logs`: Any log messages that occurred during execution of this** command, as a logging.LogRecord

Parameters

- **`protocol_file`** (*file-like*) – The protocol file to simulate.
- **`propagate_logs`** (*bool*) – Whether this function should allow logs from the Opentrons stack to propagate up to the root handler. This can be useful if you're integrating this function in a larger application, but most logs that occur during protocol simulation are best associated with the actions in the protocol that cause them.
- **`log_level`** (*'debug', 'info', 'warning', or 'error'*) – The level of logs to capture in the runlog

Returns List[Dict[str, Dict[str, Any]]] A run log for user output.

Python Module Index

O

Index