Opentrons API Documentation

Release 3.13.2

Opentrons Labworks

Oct 15, 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.

Getting Started

New to Python? Check out our *Design with Python* (page 3) page first before continuing. To get a sense of the typical structure of our scripts, take a look at our *Examples* (page 40) page.

Our API requires Python version 3.6.4 or later. Once this is set up on your computer, you can simply use *pip* to install the Opentrons package.

```
pip install opentrons
```

To simulate protocols on your laptop, check out *Simulating Your Scripts* (page 5). When you're ready to run your script on a robot, download our latest desktop app[1]

Troubleshooting

If you encounter problems using our products please take a look at our support docs[2]

or contact our team via intercom on our website at opentrons.com[3]

Feature Requests

Have an interesting idea or improvement for our software? Create a ticket on github by following these guidelines.[4]

[1] https://www.opentrons.com/ot-app
[2] https://support.opentrons.com/en/
[3] https://opentrons.com
[4] https://github.com/Opentrons/opentrons/blob/edge/CONTRIBUTING.md#opening-issues

Developer's guide

Do you want to contribute to our open-source API? You can find more information on how to be involved here.[5]

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[6]):

1. Hello World[7]
2. Variables and Types[8]
3. Lists[9]
4. Basic Operators[10]
5. Conditions[11]
6. Loops[12]
7. Functions[13]
8. Dictionaries[14]

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[15]

---

[5] https://github.com/Opentrons/opentrons/blob/edge/CONTRIBUTING.md
[6] http://www.learnpython.org/
[7] http://www.learnpython.org/en/Hello%2C_World%21
[8] http://www.learnpython.org/en/Variables_and_Types
[9] http://www.learnpython.org/en/Lists
[10] http://www.learnpython.org/en/Basic_Operators
[11] http://www.learnpython.org/en/Conditions
[12] http://www.learnpython.org/en/Loops
[13] http://www.learnpython.org/en/Functions
[14] http://www.learnpython.org/en/Dictionaries
[15] https://docs.python.org/3/index.html

Working with Python

Using a popular and free code editor, like Sublime Text 3[16], 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 5), ONLY for your locally-installed notebook) or not.

Non-Jupyter Installation

First, install Python 3.6 (Windows x64[17], Windows x86[18], OS X[19]) 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[20] 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[21] 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).

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
```

[16] https://www.sublimetext.com/3
[17] https://www.python.org/ftp/python/3.6.4/python-3.6.4-amd64.exe
[18] https://www.python.org/ftp/python/3.6.4/python-3.6.4.exe
[19] https://www.python.org/ftp/python/3.6.4/python-3.6.4-macosx10.6.pkg
[20] https://github.com/pyenv/pyenv
[21] https://pypi.org/project/opentrons/

Simulating Your Scripts

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 57) 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 57) 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 57).

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.

Opentrons API

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 your Python protocol by using the `labware.load()` method with the specific load name of the labware you need.

Please see the Opentrons Labware Library[22] for a list of currently supported labware, along with visualizations, pictures, and load names.

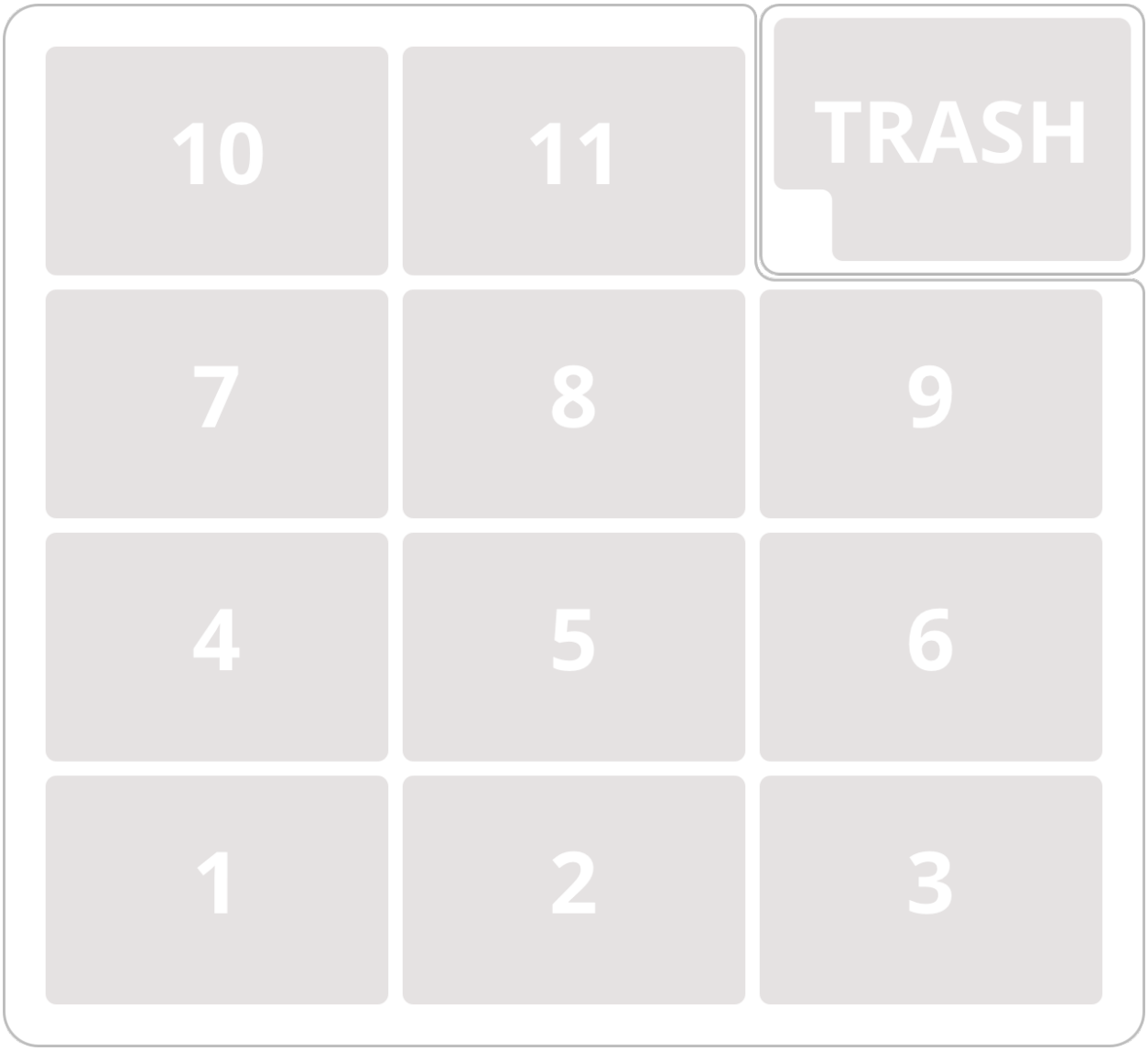
Tip: Copy and paste load names directly from the Labware Library to ensure your `load()` statements get the correct definitions.

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.

Placing labware on the robot deck

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

[22] https://labware.opentrons.com



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

# ...

tiprack = labware.load('opentrons_96_tiprack_300ul', slot='1')
```

Labware Import Reference

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

Load

`labware.load` tells the robot that your protocol will be using a given labware in a certain slot.

```
my_labware = labware.load('usascientific_12_reservoir_22ml', slot='1')
```

A third optional argument can be used to give a labware a nickname for display in the Opentrons App.

```
my_labware = labware.load('usascientific_12_reservoir_22ml',
                  slot='2',
                  label='any-name-you-want')
```

Sometimes, you may need to place a labware on top of something else on the deck, like modules. For this, you should use the `share` parameter.

```
from opentrons import labware, modules

td = modules.load('tempdeck', slot='1')
plate = labware.load('opentrons_96_aluminumblock_biorad_wellplate_200ul',
                     slot='1',
                     share=True)
```

Create

Note: The current custom labware creation mechanisms in the API are fairly limited. We're working on a much more robust system for custom labware definitions. If the current API isn't able to support your labware, please reach out to our support team.

Using `labware.create`, you can create your own custom labware. The labware created through this method must consist of circular wells arranged in regularly-spaced columns and rows.

```
custom_plate_name = 'custom_18_wellplate_200ul'

if plate_name not in labware.list():
    labware.create(
        custom_plate_name,   # name of you labware
        grid=(3, 6),         # number of (columns, rows)
        spacing=(12, 12),    # distances (mm) between each (column, row)
        diameter=5,          # diameter (mm) of each well
        depth=10,            # depth (mm) of each well
        volume=200)          # volume (µL) of each well

custom_plate = labware.load(custom_plate_name, slot='3')

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

The above example 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>
```

You only need to call `labware.create` once. It will save the labware definition on the robot so that your labware will be available to all your subsequent protocol runs.

`labware.create` **will throw an error if you try to call it more than once with the same load name**. In the example above, the call to *labware.create* is wrapped in an if-block so it does not try to add the definition twice, which would cause an error.

If you would like to delete a labware you have already added to the database (for example: to make changes to its definition), you can do the following:

```
from opentrons.data_storage import database

database.delete_container('custom_18_wellplate_200ul')
```

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

List (deprecated)

`labware.list` returns an array of all labware load names in the old, unsupported format.

```
labware.list()
```

Tip: For a list of all currently supported labware, please visit the Opentrons Labware Library[23]

Accessing Wells

Individual Wells

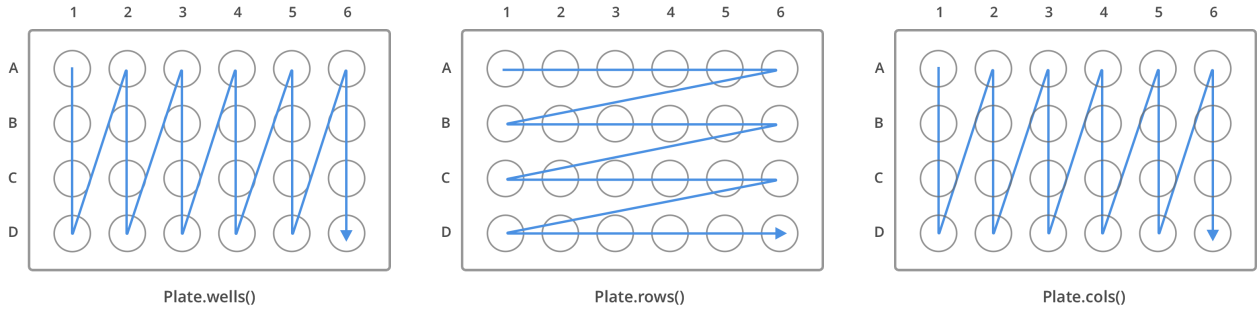
When writing a protocol using the API, you will need to select which wells to transfer liquids to and from.

The OT-2 deck and labware are all set up with the same coordinate system

- Lettered rows `['A']-['END']`

[23] https://labware.opentrons.com

- Numbered columns `['1']-['END']`.




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

plate = labware.load('corning_24_wellplate_3.4ml_flat', slot='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.

```
a1 = plate.wells('A1')
d6 = plate.wells('D6')
```

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(23)  # well D6
```

Tip: You may find well names (e.g. `B3`) to be easier to reason with, especially with irregular labware (e.g. `opentrons_10_tuberack_falcon_4x50ml_6x15ml_conical`). Whichever well access method you use, your protocol will be most maintainable if you pick one method and don't use the other one.

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 `'D'` for example) and go left to right, while columns are given numbered names (`'1'` through `'6'` 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 4 wells
Row "A" has 6 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')
```

Tip: The example above works but is a little convoluted. If you can, always get individual wells like A1 with `wells('A1')` or `wells(0)`

Multiple Wells

If we had to reference each well one at a time, our protocols could get 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 and loop (or iterate) through them.

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

plate = labware.load('corning_24_wellplate_3.4ml_flat', slot='1')
```

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', 'D4')

print(w)
```

will print out...

```
<WellSeries: <Well A1><Well B2><Well C3><Well D4>>
```

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

```
for w in plate.wells('A1', 'B2', 'C3', 'D4'):
    print(w)
```

will print out...

```
<Well A1>
<Well B2>
<Well C3>
<Well D3>
```

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='D1'):
    print(w)
```

will print out...

```
<Well A1>
<Well B1>
<Well C1>
<Well D1>
```

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('D1', to='A1'):
    print(w)
```

will print out...

```
<Well D1>
<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, including the starting well. The example below will return 4 wells, starting at well `'A1'`:

```
for w in plate.wells('A1', length=4):
    print(w)
```

will print out...

```
<Well A1>
<Well B1>
<Well C1>
<Well D1>
```

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

And here is an example of iterating through columns:

```
for c in plate.cols('1', to='6'):
    print(c)
```

will print out...

```
<WellSeries: <Well A1><Well B1><Well C1><Well D1>>
<WellSeries: <Well A2><Well B2><Well C2><Well D2>>
<WellSeries: <Well A3><Well B3><Well C3><Well D3>>
<WellSeries: <Well A4><Well B4><Well C4><Well D4>>
<WellSeries: <Well A5><Well B5><Well C5><Well D5>>
<WellSeries: <Well A6><Well B6><Well C6><Well D6>>
```

Slices

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

```
# start at index 0
# slice until index 8, without including it
# increment by 2
for w in plate[0:8:2]:
    print(w)
```

will print out...

```
<Well A1>
<Well C1>
<Well A2>
<Well C2>
```

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>
```

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

will print out...

```
<Well B1>
<Well B3>
<Well B5>
```

Deprecated Labware Load Names

Prior to version `3.10.0` of the Opentrons API, we used a completely different set of labware load names. They will continue to work until version `4.0.0` is released, but they should be considered deprecated.

We recommend you switch over to using the load names from the Labware Library as soon as possible. The following mapping can be used as a guide:

Deprecated	Recommended
6-well-plate	corning_6_wellplate_16.8ml_flat
12-well-plate	corning_12_wellplate_6.9ml_flat
24-well-plate	corning_24_wellplate_3.4ml_flat
48-well-plate	corning_48_wellplate_1.6ml_flat
384-plate	corning_384_wellplate_112ul_flat
96-deep-well	usascientific_96_wellplate_2.4ml_deep
96-flat	corning_96_wellplate_360ul_flat
96-PCR-flat	biorad_96_wellplate_200ul_pcr
96-PCR-tall	biorad_96_wellplate_200ul_pcr
alum-block-pcr-strips	opentrons_40_aluminumblock_eppendorf_24x2ml_sa
biorad-hardshell-96-PCR	biorad_96_wellplate_200ul_pcr
opentrons-aluminum-block-2ml-eppendorf	opentrons_24_aluminumblock_generic_2ml_screwca
opentrons-aluminum-block-2ml-screwcap	opentrons_24_aluminumblock_generic_2ml_screwca
opentrons-aluminum-block-96-PCR-plate	opentrons_96_aluminumblock_biorad_wellplate_20
opentrons-aluminum-block-PCR-strips-200ul	opentrons_96_aluminumblock_generic_pcr_strip_2
opentrons-tiprack-300ul	opentrons_96_tiprack_300ul
opentrons-tuberack-1.5ml-eppendorf	opentrons_24_tuberack_eppendorf_1.5ml_safelock
opentrons-tuberack-15_50ml	opentrons_10_tuberack_falcon_4x50ml_6x15ml_con
opentrons-tuberack-15ml	opentrons_15_tuberack_falcon_15ml_conical
opentrons-tuberack-2ml-eppendorf	opentrons_24_tuberack_eppendorf_2ml_safelock_s
opentrons-tuberack-2ml-screwcap	opentrons_24_tuberack_generic_2ml_screwcap
opentrons-tuberack-50ml	opentrons_6_tuberack_falcon_50ml_conical
PCR-strip-tall	opentrons_96_aluminumblock_generic_pcr_strip_2
tiprack-10ul	opentrons_96_tiprack_10ul
tiprack-200ul	tipone_96_tiprack_200ul
tiprack-1000ul	opentrons_96_tiprack_1000ul
trash-box	agilent_1_reservoir_290ml
trough-12row	usascientific_12_reservoir_22ml
tube-rack-.75ml	opentrons_24_tuberack_generic_0.75ml_snapcap_a
tube-rack-2ml	opentrons_24_tuberack_eppendorf_2ml_safelock_s
tube-rack-15_50ml	opentrons_10_tuberack_falcon_4x50ml_6x15ml_con

---

Note: If your labware is missing from the list above, or you're unsure how to update your protocol's load names, please contact our support team

---

The following load names do not have a new definitions available, and could eventually be removed. They will

continue to function normally for now. If you have any concerns about their deprecation and/or removal, please reach out!

- `24-vial-rack`
- `48-vial-plate`
- `5ml-3x4`
- `96-well-plate-20mm`
- `MALDI-plate`
- `T25-flask`
- `T75-flask`
- `e-gelgol`
- `hampton-1ml-deep-block`
- `point`
- `rigaku-compact-crystallization-plate`
- `small_vial_rack_16x45`
- `temperature-plate`
- `tiprack-10ul-H`
- `trough-12row-short`
- `trough-1row-25ml`
- `trough-1row-test`
- `tube-rack-2ml-9x9`
- `tube-rack-5ml-96`
- `tube-rack-80well`
- `wheaton_vial_rack`

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`, `dispense_speed`, and `blow_out_speed` in units of millimeters of plunger travel per second, or through `aspirate_flow_rate`, `dispense_flow_rate`, and `blow_out_flow_rate` in units of microliters/second. These have varying defaults depending on the model.

```
pipette = instruments.P300_Single(
    mount='right',
    aspirate_flow_rate=200,
    dispense_flow_rate=600,
    blow_out_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
- Blow Out Default: 1000 μl/s
- Minimum Volume: 1 μl
- Maximum Volume: 10 μl

P10_Multi

- Aspirate Default: 5 μl/s
- Dispense Default: 10 μl/s

- Blow Out Default: 1000 µl/s
- Minimum Volume: 1 µl
- Maximum Volume: 10 µl

P50_Single

- Aspirate Default: 25 µl/s
- Dispense Default: 50 µl/s
- Blow Out Default: 1000 µl/s
- Minimum Volume: 5 µl
- Maximum Volume: 50 µl

P50_Multi

- Aspirate Default: 25 µl/s
- Dispense Default: 50 µl/s
- Blow Out Default: 1000 µl/s
- Minimum Volume: 5 µl
- Maximum Volume: 50 µl

P300_Single

- Aspirate Default: 150 µl/s
- Dispense Default: 300 µl/s
- Blow Out Default: 1000 µl/s
- Minimum Volume: 30 µl
- Maximum Volume: 300 µl

P300_Multi

- Aspirate Default: 150 µl/s
- Dispense Default: 300 µl/s
- Blow Out Default: 1000 µl/s
- Minimum Volume: 30 µl
- Maximum Volume: 300 µl

P1000_Single

- Aspirate Default: 500 μl/s
- Dispense Default: 1000 μl/s
- Blow Out 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.

```
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(v_offset=-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 move 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

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

tiprack = labware.load('opentrons-tiprack-300ul', '2')

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

You could simulate the protocol using our protocol simulator, which can be installed by following the instructions [here](https://github.com/Opentrons/opentrons/tree/edge/api#simulating-protocols).

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'))
```

will have the steps...

```
Transferring 700 from well A2 in "1" to well B2 in "1"
Picking up tip well A1 in "2"
Aspirating 300.0 uL from well A2 in "1" at 1 speed
Dispensing 300.0 uL into well B2 in "1"
Aspirating 200.0 uL from well A2 in "1" at 1 speed
Dispensing 200.0 uL into well B2 in "1"
Aspirating 200.0 uL from well A2 in "1" at 1 speed
Dispensing 200.0 uL into well B2 in "1"
Dropping tip well A1 in "12"
```

Multiple Wells

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

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

will have the steps...

```
Transferring 100 from wells A1...H1 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Aspirating 100.0 uL from well B1 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well C1 in "1" at 1 speed
Dispensing 100.0 uL into well C2 in "1"
Aspirating 100.0 uL from well D1 in "1" at 1 speed
Dispensing 100.0 uL into well D2 in "1"
Aspirating 100.0 uL from well E1 in "1" at 1 speed
Dispensing 100.0 uL into well E2 in "1"
Aspirating 100.0 uL from well F1 in "1" at 1 speed
Dispensing 100.0 uL into well F2 in "1"
Aspirating 100.0 uL from well G1 in "1" at 1 speed
Dispensing 100.0 uL into well G2 in "1"
Aspirating 100.0 uL from well H1 in "1" at 1 speed
Dispensing 100.0 uL into well H2 in "1"
Dropping tip well A1 in "12"
```

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.cols('2'))
```

will have the steps...

```
Transferring 100 from well A1 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well C2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well D2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well E2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well F2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well G2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well H2 in "1"
Dropping tip well A1 in "12"
```

Few to Many

What happens if, for example, you tell your pipette to transfer from 2 source wells to 4 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'),
    plate.wells('B1', 'B2', 'B3', 'B4'))
```

will have the steps...

```
Transferring 100 from wells A1...A2 in "1" to wells B1...B4 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B3 in "1"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B4 in "1"
Dropping tip well A1 in "12"
```

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'))
```

will have the steps...

```
Transferring [20, 40, 60] from well A1 in "1" to wells B1...B3 in "1"
Picking up tip well A1 in "2"
Aspirating 20.0 uL from well A1 in "1" at 1 speed
Dispensing 20.0 uL into well B1 in "1"
Aspirating 40.0 uL from well A1 in "1" at 1 speed
Dispensing 40.0 uL into well B2 in "1"
Aspirating 60.0 uL from well A1 in "1" at 1 speed
Dispensing 60.0 uL into well B3 in "1"
Dropping tip well A1 in "12"
```

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.cols('2'))
```

will have the steps...

```
Transferring (100, 30) from well A1 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Aspirating 90.0 uL from well A1 in "1" at 1 speed
Dispensing 90.0 uL into well B2 in "1"
Aspirating 80.0 uL from well A1 in "1" at 1 speed
Dispensing 80.0 uL into well C2 in "1"
Aspirating 70.0 uL from well A1 in "1" at 1 speed
Dispensing 70.0 uL into well D2 in "1"
Aspirating 60.0 uL from well A1 in "1" at 1 speed
Dispensing 60.0 uL into well E2 in "1"
Aspirating 50.0 uL from well A1 in "1" at 1 speed
Dispensing 50.0 uL into well F2 in "1"
Aspirating 40.0 uL from well A1 in "1" at 1 speed
Dispensing 40.0 uL into well G2 in "1"
Aspirating 30.0 uL from well A1 in "1" at 1 speed
Dispensing 30.0 uL into well H2 in "1"
Dropping tip well A1 in "12"
```

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

will have the steps...

```
Consolidating 30 from wells A2...H2 in "1" to well A1 in "1"
Transferring 30 from wells A2...H2 in "1" to well A1 in "1"
Picking up tip well A1 in "2"
Aspirating 30.0 uL from well A2 in "1" at 1 speed
Aspirating 30.0 uL from well B2 in "1" at 1 speed
Aspirating 30.0 uL from well C2 in "1" at 1 speed
Aspirating 30.0 uL from well D2 in "1" at 1 speed
Aspirating 30.0 uL from well E2 in "1" at 1 speed
Aspirating 30.0 uL from well F2 in "1" at 1 speed
Aspirating 30.0 uL from well G2 in "1" at 1 speed
Aspirating 30.0 uL from well H2 in "1" at 1 speed
Dispensing 240.0 uL into well A1 in "1"
Dropping tip well A1 in "12"
```

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

```
pipette.consolidate(30, plate.cols('1'), plate.wells('A1', 'A2'))
```

will have the steps...

```
Consolidating 30 from wells A1...H1 in "1" to wells A1...A2 in "1"
Transferring 30 from wells A1...H1 in "1" to wells A1...A2 in "1"
Picking up tip well A1 in "2"
Aspirating 30.0 uL from well A1 in "1" at 1 speed
Aspirating 30.0 uL from well B1 in "1" at 1 speed
Aspirating 30.0 uL from well C1 in "1" at 1 speed
Aspirating 30.0 uL from well D1 in "1" at 1 speed
Dispensing 120.0 uL into well A1 in "1"
Aspirating 30.0 uL from well E1 in "1" at 1 speed
Aspirating 30.0 uL from well F1 in "1" at 1 speed
Aspirating 30.0 uL from well G1 in "1" at 1 speed
Aspirating 30.0 uL from well H1 in "1" at 1 speed
Dispensing 120.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

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('A'))
```

will have the steps...

```
Distributing 55 from well A1 in "1" to wells A1...A12 in "1"
Transferring 55 from well A1 in "1" to wells A1...A12 in "1"
Picking up tip well A1 in "2"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A1 in "1"
Dispensing 55.0 uL into well A2 in "1"
Dispensing 55.0 uL into well A3 in "1"
Dispensing 55.0 uL into well A4 in "1"
Blowing out at well A1 in "12"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A5 in "1"
Dispensing 55.0 uL into well A6 in "1"
Dispensing 55.0 uL into well A7 in "1"
Dispensing 55.0 uL into well A8 in "1"
Blowing out at well A1 in "12"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A9 in "1"
Dispensing 55.0 uL into well A10 in "1"
Dispensing 55.0 uL into well A11 in "1"
Dispensing 55.0 uL into well A12 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

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('A'))
```

will have the steps...

```
Distributing 30 from wells A1...A2 in "1" to wells A1...A12 in "1"
Transferring 30 from wells A1...A2 in "1" to wells A1...A12 in "1"
Picking up tip well A1 in "2"
Aspirating 210.0 uL from well A1 in "1" at 1 speed
Dispensing 30.0 uL into well A1 in "1"
Dispensing 30.0 uL into well A2 in "1"
Dispensing 30.0 uL into well A3 in "1"
Dispensing 30.0 uL into well A4 in "1"
Dispensing 30.0 uL into well A5 in "1"
Dispensing 30.0 uL into well A6 in "1"
Blowing out at well A1 in "12"
Aspirating 210.0 uL from well A2 in "1" at 1 speed
Dispensing 30.0 uL into well A7 in "1"
Dispensing 30.0 uL into well A8 in "1"
Dispensing 30.0 uL into well A9 in "1"
Dispensing 30.0 uL into well A10 in "1"
Dispensing 30.0 uL into well A11 in "1"
Dispensing 30.0 uL into well A12 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

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. There is a default disposal volume (equal to the pipette's minimum volume), which will be blown out at the trash after the dispenses.

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

will have the steps...

```
Distributing 30 from wells A1...A2 in "1" to wells A2...H2 in "1"
Transferring 30 from wells A1...A2 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 130.0 uL from well A1 in "1" at 1 speed
Dispensing 30.0 uL into well A2 in "1"
Dispensing 30.0 uL into well B2 in "1"
Dispensing 30.0 uL into well C2 in "1"
Dispensing 30.0 uL into well D2 in "1"
Blowing out at well A1 in "12"
Aspirating 130.0 uL from well A2 in "1" at 1 speed
Dispensing 30.0 uL into well E2 in "1"
Dispensing 30.0 uL into well F2 in "1"
Dispensing 30.0 uL into well G2 in "1"
Dispensing 30.0 uL into well H2 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

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
```

will have the steps...

```
Transferring 100 from wells A1...A3 in "1" to wells B1...B3 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
```

```
Dropping tip well A1 in "12"
Picking up tip well B1 in "2"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Dropping tip well A1 in "12"
Picking up tip well C1 in "2"
Aspirating 100.0 uL from well A3 in "1" at 1 speed
Dispensing 100.0 uL into well B3 in "1"
Dropping tip well A1 in "12"
```

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

will have the steps...

```
Picking up tip well A1 in "2"
...
Transferring 100 from wells A1...A3 in "1" to wells B1...B3 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well A3 in "1" at 1 speed
Dispensing 100.0 uL into well B3 in "1"
...
Dropping tip well A1 in "12"
```

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
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well B1 in "1"
Picking up tip well A1 in "2"
```

```
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Returning tip
Dropping tip well A1 in "2"
```

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
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Touching tip
Dispensing 100.0 uL into well A2 in "1"
Touching tip
Dropping tip well A1 in "12"
```

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
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Blowing out
Dropping tip well A1 in "12"
```

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
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Mixing 2 times with a volume of 50ul
Aspirating 50 uL from well A1 in "1" at 1.0 speed
Dispensing 50 uL into well A1 in "1"
Aspirating 50 uL from well A1 in "1" at 1.0 speed
Dispensing 50 uL into well A1 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Mixing 3 times with a volume of 75ul
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

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
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Air gap
Aspirating 20 uL from well A1 in "1" at 1.0 speed
Dispensing 20 uL into well A2 in "1"
Dispensing 100.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

```
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[24].

```
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
```

[24] https://github.com/Opentrons/opentrons/blob/edge/api/src/opentrons/config/robot_configs.py

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[25].

```
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')

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...

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

```
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[26].

```
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.

```
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

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

```
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 using *Placeable.from_center()* (page 55) to specify locations within 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 57).

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*, *model=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*, *blow_out_speed=60*, *aspirate_flow_rate=None*, *dispense_flow_rate=None*, *plunger_current=0.5*, *drop_tip_current=0.5*, *return_tip_height=None*, *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*, *blow_out_flow_rate=None*)

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 47), *dispense()* (page 48), *mix()* (page 50), and *blow_out()* (page 48)
- Handle tips with *pick_up_tip()* (page 51), *drop_tip()* (page 50), and *return_tip()* (page 52)
- 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 50) 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 51) (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 46).

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* (page 55) or tuple(*Placeable* (page 55), `Vector`)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the aspirate. Can also be a tuple with first item *Placeable* (page 55), 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 46).

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* (page 55) or tuple(*Placeable* (page 55), Vector)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the blow_out. Can also be a tuple with first item *Placeable* (page 55), second item relative Vector

Returns

Return type This instance of *Pipette* (page 46).

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 46).

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* (page 55) or tuple(*Placeable* (page 55), `Vector`)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the dispense. Can also be a tuple with first item *Placeable* (page 55), 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 46).

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 46).

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 46))

Parameters **location** (*Placeable* (page 55) or tuple(*Placeable* (page 55), Vector)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the drop_tip. Can also be a tuple with first item *Placeable* (page 55), second item relative Vector

Returns

Return type This instance of *Pipette* (page 46).

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 46).

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* (page 55) or tuple(*Placeable* (page 55), `Vector`)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the mix. Can also be a tuple with first item *Placeable* (page 55), 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 46).

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 46) to a *Placeable* (page 55) on the `Deck`

Notes

Until obstacle-avoidance algorithms are in place, *Robot* (page 43) and *Pipette* (page 46) *move_to()* (page 51) use either an "arc" or "direct"

Parameters

- **location** (*Placeable* (page 55) or tuple(*Placeable* (page 55), `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 46).

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 46))

Parameters

- **location** (*Placeable* (page 55) or tuple(*Placeable* (page 55), Vector)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the pick_up_tip. Can also be a tuple with first item *Placeable* (page 55), 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 46).

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 46))

Returns

Return type This instance of *Pipette* (page 46).

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*, *blow_out=None*)

Set the speed (uL/second) the *Pipette* (page 46) plunger will move during *aspirate()* (page 47) and *dispense()* (page 48). 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 46) 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* (page 55) or tuple(*Placeable* (page 55), Vector)) – The *Placeable* (page 55) (*Well* (page 111)) to perform the touch_tip. Can also be a tuple with first item *Placeable* (page 55), 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 53) will move to 100% of the wells radius. When radius=0.5, *touch_tip()* (page 53) 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 46).

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 46) commands, like *aspirate* (page 47) and *dispense* (page 48), 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* (page 55) and add to each *targets* *Placeable* (page 55). 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* (page 55) or list of :any:‘Placeable‘s, from where liquid will be :any:‘aspirate‘ed from.
- **dest** (*Placeable or list*) – Single *Placeable* (page 55) 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 *True* (default behavior) and trash container has been attached to this *Pipette*, then the tip will be sent to the trash container. If *False*, then tips will be returned to their associated tiprack.
- **touch_tip** (*boolean*) – If *True*, a *touch_tip* (page 53) will occur following each *aspirate* (page 47) and *dispense* (page 48). If set to *False* (default), no *touch_tip* (page 53) will occur.
- **blow_out** (*boolean*) – If *True*, a *blow_out* (page 48) will occur following each *dispense* (page 48), but only if the pipette has no liquid left in it. If set to *False* (default), no *blow_out* (page 48) will occur.
- **mix_before** (*tuple*) – Specify the number of repetitions volume to mix, and a *Mix* (page 80) will proceed each *aspirate* (page 47) 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 80) will following each *dispense* (page 48) 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 49) and *consolidate* (page 48))If *True* (default), sequential *aspirate* (page 47) 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 46).

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

Placeable

class opentrons.legacy_api.containers.placeable.**Placeable**(*parent=None*, *properties=None*)

This class represents every item on the deck.

It maintains the hierarchy and provides means to: * traverse * retrieve items by name * calculate coordinates in different reference systems

It should never be directly created; it is created by the system during labware load and when accessing wells.

bottom(*z=0*, *radius=0*, *degrees=0*, *reference=None*)

Returns (*Placeable* (page 55), Vector) tuple where the vector points to the bottom of the placeable. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

If `reference` (a *Placeable* (page 55)) is provided, the return value will be in that placeable's coordinate system.

The `radius` and `degrees` arguments are interpreted as in *from_center()* (page 55) (except that `degrees` is in degrees, not radians). They can be used to specify a further distance from the bottom center of the well; for instance, calling `bottom(radius=0.5,degrees=180)` will move half the radius in the 180 degree direction from the center of the well.

The `z` argument is a distance in mm to move in z from the bottom, and can be used to hover above the bottom. For instance, calling `bottom(z=1)` will move 1mm above the bottom.

Parameters

- **z** – Absolute distance in mm to move in `z` from the bottom. Note that unlike the other arguments, this is a distance, not a ratio.
- **degrees** – Direction in which to move `radius` from the bottom center.
- **radius** – Ratio of the placeable's radius to move in the direction specified by `degrees` from the bottom center.
- **reference** – An optional placeable for the vector to be relative to.

Returns A tuple of the placeable and the offset. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

center(*reference=None*)

Returns (*Placeable* (page 55), Vector) tuple where the vector points to the center of the placeable, in `x`, `y`, and `z`. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

If `reference` (a *Placeable* (page 55)) is provided, the return value will be in that placeable's coordinate system.

Parameters **reference** – An optional placeable for the vector to be relative to.

Returns A tuple of the placeable and the offset. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

from_center(*x=None*, *y=None*, *z=None*, *r=None*, *theta=None*, *h=None*, *reference=None*)

Accepts a set of ratios for Cartesian or ratios/angle for Polar and returns `Vector` using `reference` as origin.

Though both polar and cartesian arguments are accepted, only one set should be used at the same time, and the set selected should be entirely used. In addition, all variables in the set should be used.

For instance, if you want to use cartesian coordinates, you must specify all of `x`, `y`, and `z` as numbers; if you want to use polar coordinates, you must specify all of `theta`, `r` and `h` as numbers.

While `theta` is an absolute angle in radians, the other values are actually ratios which are multiplied by the relevant dimensions of the placeable on which `from_center` is called. For instance, calling `from_center(x=0.5,y=0.5,z=0.5)` does not mean "500 micromenters from the center in each dimension", but "half the x size, half the y size, and half the z size from the center". Similarly, `from_center(r=0.5,theta=3.14,h=0.5)` means "half the radius dimension at 180 degrees, and half the height upwards".

Parameters

- **x** – Ratio of the x dimension of the placeable to move from the center.
- **y** – Ratio of the y dimension of the placeable to move from the center.
- **z** – Ratio of the z dimension of the placeable to move from the center.
- **r** – Ratio of the radius to move from the center.
- **theta** – Angle in radians at which to move the percentage of the radius specified by `r` from the center.
- **h** – Percentage of the height to move up in z from the center.
- **reference** – If specified, an origin to add to the offset vector specified by the other arguments.

Returns A vector from either the origin or the specified reference. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

top(*z=0*, *radius=0*, *degrees=0*, *reference=None*)

Returns (*Placeable* (page 55), `Vector`) tuple where the vector points to the top of the placeable. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

If `reference` (a *Placeable* (page 55)) is provided, the return value will be in that placeable's coordinate system.

The `radius` and `degrees` arguments are interpreted as in *from_center()* (page 55) (except that `degrees` is in degrees, not radians). They can be used to specify a further distance from the top center of the well; for instance, calling `top(radius=0.5,degrees=180)` will move half the radius in the 180 degree direction from the center of the well.

The `z` argument is a distance in mm to move in z from the top, and can be used to hover above or below the top. For instance, calling `top(z=-1)` will move 1mm below the top.

Parameters

- **z** – Absolute distance in mm to move in `z` from the top. Note that unlike the other arguments, this is a distance, not a ratio.
- **degrees** – Direction in which to move `radius` from the top center.
- **radius** – Ratio of the placeable's radius to move in the direction specified by `degrees` from the top center.

Returns A tuple of the placeable and the offset. This can be passed into any *Robot* (page 43) or *Pipette* (page 46) method `location` argument.

Simulation

opentrons.simulate: functions and entrypoints for simulating protocols

This module has functions that provide a console entrypoint for simulating a protocol from the command line.

`opentrons.simulate.`**`bundle_from_sim`**(*protocol: opentrons.protocols.types.PythonProtocol*, *context: opentrons.protocol_api.contexts.ProtocolContext*) → opentrons.protocols.types.BundleContents

From a protocol, and the context that has finished simulating that protocol, determine what needs to go in a bundle for the protocol.

`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 57)

`opentrons.simulate.`**`get_arguments`**(*parser: argparse.ArgumentParser*) → argparse.ArgumentParser

Get the argument parser for this module

Useful if you want to use this module as a component of another CLI program and want to add its arguments.

Parameters **`parser`** – A parser to add arguments to. If not specified, one will be created.

Returns argparse.ArgumentParser The parser with arguments added.

`opentrons.simulate.`**`simulate`**(*protocol_file: typing.TextIO*, *file_name: str*, *custom_labware_paths=None*, *custom_data_paths=None*, *propagate_logs=False*, *log_level='warning'*) → typing.Tuple[typing.List[typing.Mapping[str, typing.Any]], typing.Union[opentrons.protocols.types.BundleContents, NoneType]]

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; and either the contents of the protocol that would be required to bundle, or `None`.

Each dict element in the run log 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.
- **file_name** (*str*) – The name of the file
- **custom_labware_paths** – A list of directories to search for custom labware, or None. Ignored if the apiv2 feature flag is not set. Loads valid labware from these paths and makes them available to the protocol context.
- **custom_data_paths** – A list of directories or files to load custom data files from. Ignored if the apiv2 feature flag if not set. Entries may be either files or directories. Specified files and the non-recursive contents of specified directories are presented by the protocol context in *ProtocolContext.bundled_data* (page 95).
- **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. Default: `False`
- **log_level** (*'debug', 'info', 'warning', or 'error'*) – The level of logs to capture in the runlog. Default: `'warning'`

Returns A tuple of a run log for user output, and possibly the required data to write to a bundle to bundle this protocol. The bundle is only emitted if the API v2 feature flag is set and this is an unbundled python protocol. In other cases it is None.

Overview

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'))
```

How it's Organized

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

1. *Imports* (page 59)
2. *Metadata* (page 59)
3. *Labware* (page 59)
4. *Pipettes* (page 60)
5. *Commands* (page 60)

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
```

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.

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')
```

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])
```

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'))
```

Opentrons API Version 2

Pipettes

Pipettes are created and attached to a specific mount on the OT-2 (`'left'` or `'right'`) on the OT-2 using the function *`ProtocolContext.load_instrument()`* (page 96) . from the `ProtocolContext` class. This will return an `InstrumentContext` object. See *Building Block Commands* (page 76) and *Complex Commands* (page 82) for liquid handling commands from the `InstrumentContext` class.

This section discusses the details of creating a pipette, and the behaviors of the pipette you can alter.

Creating A Pipette

Pipettes are specified in a protocol using the method *`ProtocolContext.load_instrument()`* (page 96). This method requires the model of the instrument to load, the mount to load it in, and (optionally) a list of associated tipracks:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    # Load a P50 multi on the left slot
    left = protocol.load_instrument('p50_multi', 'left')
    # Load a P1000 Single on the right slot, with two racks of tips
    tiprack1 = protocol.load_labware('opentrons_96_tiprack_1000ul', 1)
    tiprack2 = protocol.load_labware('opentrons_96_tiprack_1000ul', 2)
```

```
    right = protocol.load_instrument('p1000_single', 'right',
                                     tip_racks=[tiprack1, tiprack2])
```

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:

Pipette Type	Model Name
`P10 Single` (1 - 10 ul)	`'p10_single'`
`P10 Multi` (1 - 10 ul)	`'p10_multi'`
`P50 Single` (5 - 50 ul)	`'p50_single'`
`P50 Multi` (5 - 50 ul)	`'p50_multi'`
`P300 Single` (30 - 300 ul)	`'p300_single'`
`P300 Multi` (30 - 300 ul)	`'p300_multi'`
`P1000 Single` (100 - 1000 ul)	`'p1000_single'`

For every pipette type you are using in a protocol, you must use one of the model names specified above.

Adding Tip Racks

`ProtocolContext.load_instrument()` (page 96) has one important optional parameter: `tipracks`. This parameter accepts a *list* of tiprack labware objects, allowing you to specify as many tipracks as you want. Associating tipracks with your pipette allows for automatic tip tracking throughout your protocol, which removes the need for specifying tip locations in *`InstrumentContext.pick_up_tip()`* (page 105).

For instance, in this protocol you can see the effects of specifying tipracks:

This is further discussed in *Building Block Commands* (page 76) and *Complex Commands* (page 82).

Modifying Pipette Behaviors

The OT-2 has many default behaviors that are occasionally appropriate to change for a particular experiment or liquid. This section details those behaviors.

Plunger Flow Rates

Opentrons pipettes have different rates of aspiration and dispense, depending on internal mechanical details. In general, you should not increase aspiration and dispense flow rates above their defaults; however, some experiments and protocols require slower rates of aspiration and dispense. These flow rates can be changed on a created *`InstrumentContext`* (page 99) at any time, in units of microliters/sec by altering *`InstrumentContext.flow_rate`* (page 103). This has the following attributes:

- `InstrumentContext.flow_rate.aspirate`: The aspirate flow rate, in ul/s
- `InstrumentContext.flow_rate.dispense`: The dispense flow rate, in ul/s
- `InstrumentContext.flow_rate.blow_out`: The blow out flow rate, in ul/s

Each of these attributes can be altered without affecting the others.

`InstrumentContext.speed` (page 105) offers the same functionality, but controlled in units of mm/s of plunger speed. This does not have a linear transfer to flow rate and should only be used if you have a specific need.

Default Positions Within Wells

By default, the OT-2 will aspirate and dispense 1mm above the bottom of a well. This may not be suitable for some labware and well geometries, liquids, or experimental protocols. While you can specify the exact location within a well in direct calls to *InstrumentContext.aspirate()* (page 100) and *InstrumentContext.dispense()* (page 102) (see the *Specifying Position Within Wells* (page 67) section), you cannot use this method in complex commands like *InstrumentContext.transfer()* (page 106), and it can be cumbersome to specify the position every time.

Instead, you can use the attribute *InstrumentContext.well_bottom_clearance* (page 107) to specify the height above the bottom of a well to either aspirate or dispense:

1. Editing `pipette.well_bottom_clearance.aspirate` changes the height of aspiration
2. Editing `pipette.well_bottom_clearance.dispense` changes the height of dispense

Changing these attributes will affect *all* aspirates and dispenses, even those executed as part of a transfer.

```
from opentrons import protocol_api, types

def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', '1')
    pipette = protocol.load_instrument('p300_single', 'right')
    plate = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    pipette.pick_up_tip()
    # Aspirate 1mm above the bottom of the well
    pipette.aspirate(50, plate['A1'])
    # Dispense 1mm above the bottom of the well
    pipette.dispense(50, plate['A1'])
    # Aspirate 2mm above the bottom of the well
    pipette.well_bottom_clearance.aspirate = 2
    pipette.aspirate(50, plate['A1'])
    # Still dispensing 1mm above the bottom
    pipette.dispense(50, plate['A1'])
    pipette.aspirate(50, plate['A1'])
    # Dispense high above the well
    pipette.well_bottom_clearance.dispense = 10
    pipette.dispense(50, plate['A1'])
```

Head Speed

The OT-2's gantry usually moves as fast as it can given its construction; this makes protocol execution faster and saves time. However, some experiments or liquids may value slower, gentler movements over protocol execution time. In this case, you can alter the OT-2 gantry's speed when a specific pipette is moving by setting *InstrumentContext.default_speed* (page 101). This is a value in mm/s that controls the overall speed of the gantry. Its default is 400 mm/s.

Warning: The default of 400 mm/s was chosen because it is the maximum speed Opentrons knows will work with the gantry. Your specific robot may be able to move faster, but you shouldn't make this value higher than the default without extensive experimentation.

```
from opentrons import protocol_api, types

def run(protocol: protocol_api.ProtocolContext):
```

```
    pipette = protocol.load_instrument('p300_single', 'right')
    # Move to 50mm above the front left of slot 5, very quickly
    pipette.move_to(protocol.deck.position_for('5').move(types.Point(z=50)))
    # Slow down the pipette
    pipette.default_speed = 100
    # Move to 50mm above the front left of slot 9, much more slowly
    pipette.move_to(protocol.deck.position_for('9').move(types.Point(z=50)))
```

Defaults

Head Speed: 400 mm/s

Well Bottom Clearances

- Aspirate default: 1mm above the bottom
- Dispense default: 1mm above the bottom

p10_single

- Aspirate Default: 5 μl/s
- Dispense Default: 10 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 1 μl
- Maximum Volume: 10 μl

p10_multi

- Aspirate Default: 5 μl/s
- Dispense Default: 10 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 1 μl
- Maximum Volume: 10 μl

p50_single

- Aspirate Default: 25 μl/s
- Dispense Default: 50 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 5 μl
- Maximum Volume: 50 μl

p50_multi

- Aspirate Default: 25 μl/s
- Dispense Default: 50 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 5 μl
- Maximum Volume: 50 μl

p300_single

- Aspirate Default: 150 μl/s
- Dispense Default: 300 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 30 μl
- Maximum Volume: 300 μl

p300_multi

- Aspirate Default: 150 μl/s
- Dispense Default: 300 μl/s
- Blow Out Default: 1000 μl/s
- Minimum Volume: 30 μl
- Maximum Volume: 300 μl

p1000_single

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

Labware

The labware section informs the protocol context what labware is present on the robot's deck. In this section, you define the tip racks, well plates, troughs, tubes, or anything else you've put on the deck.

Each labware is given a name (e.g. `'corning_96_wellplate_360ul_flat'`), and the slot on the robot it will be placed (e.g. `'2'`). The first place to look for the names of labware should always be the Opentrons Labware Library[27], where Opentrons maintains a database of labwares, their load names, what they look like, manufacturer part numbers, and more. In this example, we'll use `'corning_96_wellplate_360ul_flat'` (an ANSI standard 96-well plate[28]) and `'opentrons_96_tiprack_300ul'` (the Opentrons standard 300 µL tiprack[29]).

From the example given on the home page, the "labware" section looked like:

```
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', '2')
tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', '1')
```

and informed the protocol context that the deck contains a 300 µL tiprack in slot 1 and a 96 well plate in slot 2.

Labware is loaded into a protocol using *`ProtocolContext.load_labware()`* (page 97), which returns a *`opentrons.protocol_api.labware.Labware`* (page 108) object. You'll never create one of these objects directly, only store them in variables from the return value of *`ProtocolContext.load_labware()`* (page 97).

[27] https://labware.opentrons.com
[28] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[29] https://labware.opentrons.com/opentrons_96_tiprack_300ul

Accessing Wells in Labware

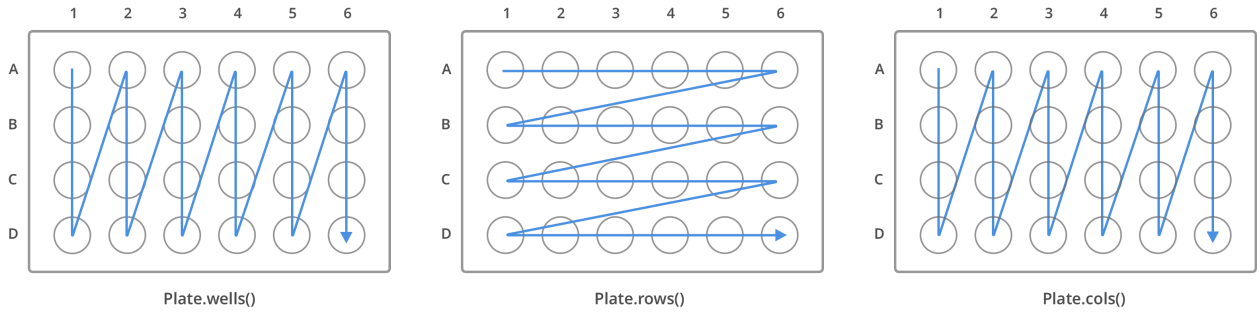
Well Ordering

When writing a protocol using the API, you will need to select which wells to transfer liquids to and from.

Rows of wells (see image below) on a labware are typically labeled with capital letters starting with `'A'`; for instance, an 8x12 96 well plate will have rows `'A'` through `'H'`.

Columns of wells (see image below) on a labware are typically labeled with numerical indices starting with `'1'`; for instance, an 8x12 96 well plate will have columns `'1'` through `'12'`.

For all well accessing functions, the starting well will always be at the top left corner of the labware. The ending well will be in the bottom right, see the diagram below for further explanation.




```
'''
Examples in this section expect the following
'''
def run(protocol):

    plate = protocol.load_labware('corning_24_wellplate_3.4ml_flat', slot='1')
```

Accessor Methods

As part of API Version 2, we wanted to allow users to utilize python's data structures more easily and as intended. That is why all of our labware accessor methods return either a dictionary, list or an individual Well object.

The table below lists out the different methods available to you and their differences.

Method Name	Returns
Labware.wells() (page 110)	List of all wells, i.e. `[labware:A1,labware:B1,labware:C1...]`
Labware.rows() (page 110)	List of a list ordered by row, i.e `[[labware:A1,labware:A2...],[labware:B1,labware:B2..]]`
Labware.columns() (page 108)	List of a list ordered by column, i.e. `[[labware:A1,labware:B1..],[labware:A2,labware:B2..]]`
Labware.wells_by_name() (page 110)	Dictionary with well names as keys, i.e. `{'A1': labware:A1,'B1': labware:B1}`
Labware.rows_by_name() (page 110)	Dictionary with row names as keys, i.e. `{'A': [labware:A1,labware:A2..],'B': [labware:B1,labware:B2]}`
Labware.columns_by_name() (page 108)	Dictionary with column names as keys, i.e. `{'1': [labware:A1,labware:B1..],'2': [labware:A2,labware:B2..]}`

Accessing Individual Wells

Dictionary Access

Once a labware is loaded into your protocol, you can easily access the many wells within it by using dictionary indexing. If a well does not exist in this labware, you will receive a `KeyError`. This is equivalent to using the return value of *`Labware.wells_by_name()`* (page 110):

```
a1 = plate['A1']
d6 = plate.wells_by_name()['D6']
```

List Access From `wells`

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()[23]  # well D6
```

Tip: You may find well names (e.g. `B3`) to be easier to reason with, especially with irregular labware (e.g. `opentrons_10_tuberack_falcon_4x50ml_6x15ml_conical` (Labware Library[30]). Whichever well access method you use, your protocol will be most maintainable if you pick one method and don't use the other one.

Accessing Groups of Wells

If we had to reference each well one at a time, our protocols could get 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 and loop (or iterate) through them.

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 `'D'` for example) and go left to right, while columns are given numbered names (`'1'` through `'6'` for example) and go from front to back.

You can access a specific row or column by using the *`Labware.rows_by_name()`* (page 110) and *`Labware.columns_by_name()`* (page 108) methods on a labware. These methods both return a dictionary with the row or column name as the index:

```
row_dict = plate.rows_by_name()['A']
row_list = plate.rows()[0] # equivalent to the line above
column_dict = plate.columns_by_name()['1']
column_list = plate.columns()[0] # equivalent to the line above

print('Column "1" has', len(column_dict), 'wells')
print('Row "A" has', len(row_dict), 'wells')
```

will print out...

```
Column "1" has 4 wells
Row "A" has 6 wells
```

[30] https://labware.opentrons.com/opentrons_10_tuberack_falcon_4x50ml_6x15ml_conical

So, since our methods return either lists or dictionaries, you can iterate through them as you would regular python data structures.

For example, if I wanted to access the individual wells of row 'A' in my well plate, I could simply do:

```
for well in plate.rows()[0]:
    print(well)
```

or,

```
for well_obj in plate.rows_by_name()['A'].values():
    print(well_obj)
```

and it will return the individual well objects in row A.

Specifying Position Within Wells

The functions listed above (in the *Accessing Wells in Labware* (page 65) section) return objects (or lists, lists of lists, dictionaries, or dictionaries of lists of objects) representing wells. These are *opentrons.protocol_api.labware.Well* (page 111) objects. Similar to the *Labware* (page 108) objects, you'll never create one of these directly - only handle them as the return values of various methods. *Well* (page 111) objects have some useful methods on them, however, which allow you to more closely specify the location to which the robot should move *inside* a given well.

Each of these methods returns an object called a *opentrons.types.Location* (page 118), which encapsulates a position in deck coordinates (see *Deck Coordinates* (page 119)) and a well with which it is associated. This lets you do further manipulations on the positions returned by these methods. All *InstrumentContext* (page 99) methods that involve positions accept these *Location* (page 118) objects.

Position Modifiers

Top

The method *Well.top()* (page 111) returns a position at the top center of the well. This is a good position to use for *Blow Out* (page 80) or any other operation where you don't want to be contacting the liquid. In addition, *Well.top()* (page 111) takes an optional argument `z`, which is a distance in mm to move relative to the top vertically (positive numbers move up, and negative numbers move down):

```
plate['A1'].top()      # This is the top center of the well
plate['A1'].top(z=1)   # This is 1mm above the top center of the well
plate['A1'].top(z=-1)  # This is 1mm below the top center of the well
```

Bottom

The method *Well.bottom()* (page 111) returns a position at the bottom center of the well. This could be a good position to start at when considering where to aspirate, or any other operation where you want to be contacting the liquid. In addition, *Well.bottom()* (page 111) takes an optional argument `z`, which is a distance in mm to move relative to the bottom vertically (positive numbers move up, and negative numbers move down):

```
plate['A1'].bottom()      # This is the bottom center of the well
plate['A1'].bottom(z=1)   # This is 1mm above the bottom center of the well
plate['A1'].bottom(z=-1)  # This is 1mm below the bottom center of the well.
                          # this may be dangerous!
```

Warning: Negative `z` arguments to *`Well.bottom()`* (page 111) may cause the tip to collide with the bottom of the well. The OT-2 has no sensors to detect this, and if it happens, the robot will be too high in z for the rest of the protocol.

Note: If you are using this to change the position at which the robot does *Aspirate* (page 79) or *Dispense* (page 79) throughout the protocol, consider setting the default aspirate or dispense offset with *`InstrumentContext.well_bottom_clearance`* (page 107) (see *Default Positions Within Wells* (page 62)).

Center

The method *`Well.center()`* (page 111) returns a position centered in the well both vertically and horizontally. This can be a good place to start for precise control of positions within the well for unusual or custom labware.

```
plate['A1'].center() # This is the vertical and horizontal center of the well
```

Manipulating Positions

The objects returned by the position modifier functions are all instances of *`opentrons.types.Location`* (page 118), which are named tuples[31] representing the combination of a point in space (another named tuple) and a reference to the associated *`Well`* (page 111) (or *`Labware`* (page 108), or slot name, depending on context).

To further change positions, you can use *`Location.move()`* (page 118), which lets you move the Location. This function takes a single argument, `point`, which should be a *`opentrons.types.Point`* (page 119). This is a named tuple with elements `x`, `y`, and `z`, representing a 3 dimensional point.

To move a location, you create a *`types.Point`* (page 119) representing a 3d offset and give it to *`Location.move()`* (page 118):

```
from opentrons import types

def run(protocol):
    plate = protocol.load_labware(
        'corning_24_wellplate_3.4ml_flat', slot='1')
    plate['A1'].center().move(
        types.Point(x=1, y=1, z=1)) # 1mm up, to the right, and towards the
                                    # back of the robot
```

‘

Hardware Modules

Modules are peripherals that attach to the OT-2 to extend its capabilities.

Modules currently supported are the Temperature, Magnetic and Thermocycler Module.

This is not an exhaustive list of functionality for modules. Check *Modules* (page 114) or on our github for full explanations of methods.

[31] https://docs.python.org/3/library/collections.html#collections.namedtuple

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 the following *inside* the run function:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
     module = protocol.load_module('Module Name', slot)
```

You can reference your module in a few different ways. The valid names can be found below. They are not case-sensitive.

Module Type	Nickname(s)
`Temperature Module`	`'Temperature Module'`, `'tempdeck'`
`Magnetic Module`	`'Magnetic Module'`, `'magdeck'`
`Thermocycler Module`	`'Thermocycler Module'`, `'thermocycler'`

Loading Labware onto your Module

If you want to interact with labware on top of your Module, you must load labware onto the module. You can do this via:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
     module = protocol.load_module('Module Name', slot)
     my_labware = module.load_labware('labware_definition_name')
```

Where `module` is the variable name you saved your module to. You do not need to specify the slot.

Checking the status of your Module

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

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
     module = protocol.load_module('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'`. For the Thermocycler Module this will return `'holding at target'` or `'idle'`. There are more detailed status checks which can be found in at *Thermocycler Module* (page 71)

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. For the purposes of this section, assume we have the following already:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    temp_mod = protocol.load_module('temperature module', '1')
    plate = temp_mod.load_labware('corning_96_wellplate_360ul_flat')
    # The code from the rest of the examples in this section goes here
```

Set Temperature

To set the temperature module to 4 degrees celsius do the following:

```
temp_mod.set_temperature(4)
```

Wait Until Setpoint Reached

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

```
temp_mod.set_temperature(4)
temp_mod.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:

```
temp_mod.temperature
```

Read the Target Temperature

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

```
temp_mod.target
```

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.

```
temp_mod.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

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.

The magnetic module has the following methods that can be accessed during a protocol. For the purposes of this section, assume we have the following already:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    mag_mod = protocol.load_module('magnetic module', '1')
    plate = mag_mod.load_labware('nest_96_wellplate_100ul_pcr_full_skirt')
    # The code from the rest of the examples in this section goes here
```

Engage

The destination of the magnets can be specified in several different ways, based on internally stored default heights for labware:

- If neither `height` nor `offset` is specified **and** the labware is support on the magnetic module, the magnets will raise to a reasonable default height based on the specified labware.

```
mag_mod.engage()
```

- If `height` is specified, it should be a distance in mm from the home position of the magnets.

```
mag_mod.engage(height=18.5)
```

Note Only certain labwares have defined engage heights for the Magnetic Module. If a labware that does not have a defined engage height is loaded on the Magnetic Module (or if no labware is loaded), then `height` must be specified.

Disengage

```
mag_mod.disengage()
```

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

Thermocycler Module

The Thermocycler Module is still under active development. The commands are subject to change. A valid operational range has not been determined yet.

The Thermocycler Module allows users to perform complete experiments that require temperature sensitive reactions such as PCR, restriction enzyme etc. Below is a description of a few ways you can control this module.

There are two heating mechanisms in the Thermocycler module which the user has access to. One is the bottom plate (*block*) in which samples are located, the other is the lid heating pad.

For the purposes of this section, assume we have the following already:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    tc_mod = protocol.load_module('Thermocycler Module')
    plate = tc_mod.load_labware('nest_96_wellplate_100ul_pcr_full_skirt')
```

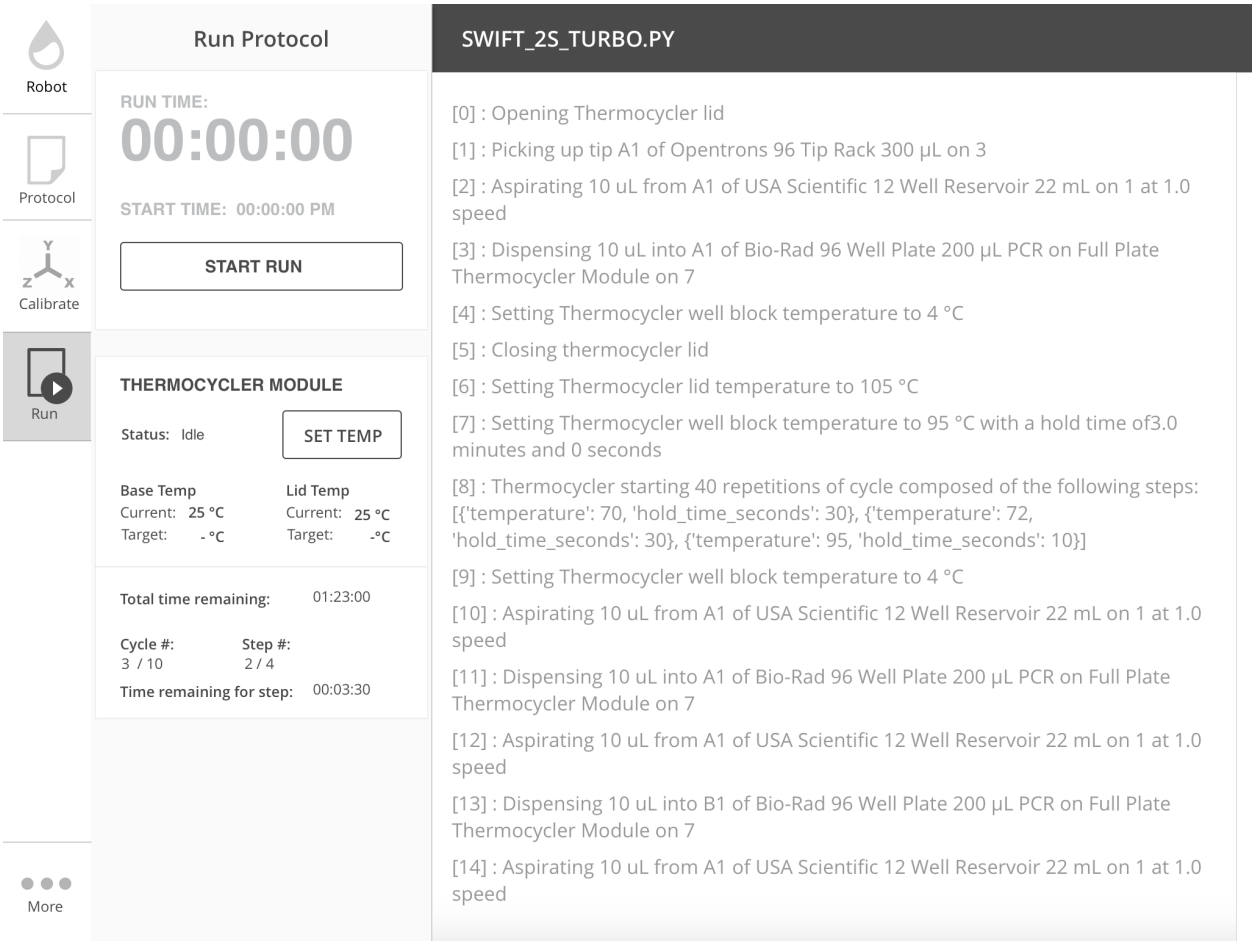
Note: When loading the Thermocycler Module, it is not necessary to specify a slot. This is because the Thermocycler Module has a default position that covers Slots 7, 8, 10, and 11. This is the only valid location for the Thermocycler Module on the OT2 deck.

Run App Control

Certain functionality of the Thermocycler Module can be controlled in the Opentrons App.

Setting a Target Temperature

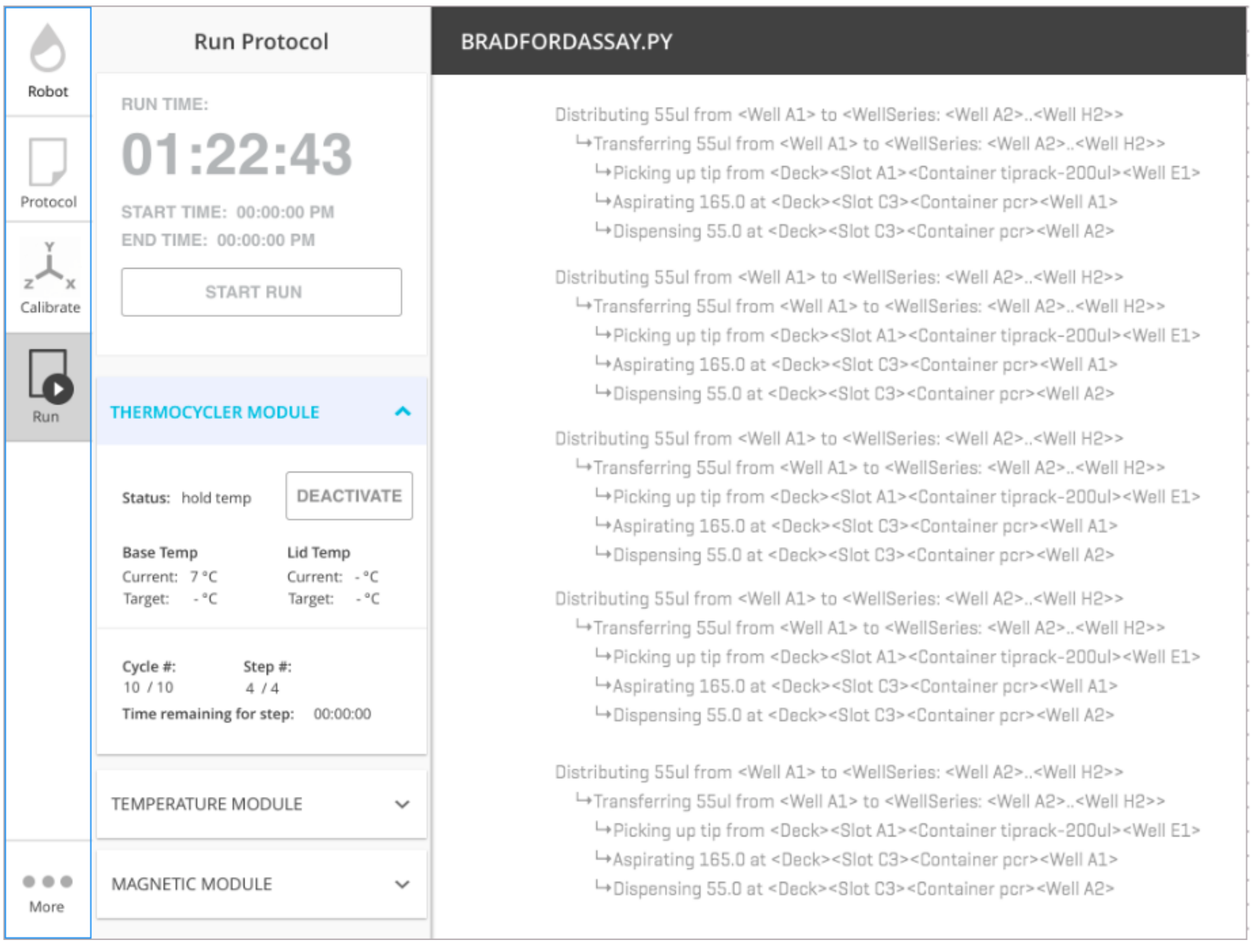
Before the run of the protocol, when you navigate to the *Run* tab of the Opentrons App, you will see a Thermocycler Module card on the left-hand side like the image below.



If you wish to set a target temperature for the Thermocycler Module *block* before a protocol run, you may do so. When you run your actual protocol, the steps will not proceed until the target temperature that was set is reached. We recommend using this if you want to pre-heat or pre-cool samples located on your Thermocycler Module.

Deactivating the Module

Sometimes you may wish to deactivate the Thermocycler Module, such as removing samples from the module or shutting the module off after use. Before or after a protocol run, you can press *deactivate* to ensure that your Thermocycler Module is off before opening the lid.



Note: The thermocycler will hold at whatever temperature it was last told to hold at (whether through the Opentrons App or through the protocol), regardless of the status of an ongoing (or not) run. This allows you to cancel a run and be sure that your samples will be held at the temperature specified previously, until you decide to deactivate the module from the Opentrons App as described above.

Lid Motor Control

The Thermocycler Module supports temperature control with the lid open and closed. When the lid of the Thermocycler Module is open, the pipettes can access the contained 96-well labware. You can control the lid with the methods below.

Open Lid

```
tc_mod.open_lid()
```

Close Lid

```
tc_mod.close_lid()
```

Lid Temperature Control

As mentioned before, users have access to controlling when a lid temperature is set. It is recommended that you set the lid temperature before executing a Thermocycler Module profile, described later. The range of the Thermocycler Module lid is 37°C to 110°C.

Set Lid Temperature

ThermocyclerContext.set_lid_temperature() (page 118) takes in one parameter which is the temperature you wish the lid to be set to. The protocol will only proceed once the lid temperature has been reached.

```
tc_mod.set_lid_temperature(temperature)
```

Block Temperature Control

To set the aluminum block temperature inside the Thermocycler Module, you can use the method *ThermocyclerContext.set_block_temperature()* (page 118). It takes in four parameters `temperature`, `hold_time_seconds`, `hold_time_minutes` and `ramp_rate` respectively. Only temperature is required, both the hold time parameters and ramp rate are not required.

Temperature

If you only specify a temperature in celsius, the Thermocycler Module will hold this temperature indefinitely until powered off.

```
tc_mod.set_block_temperature(4)
```

Hold Time

If you set a temperature and a hold time, the Thermocycler Module will hold the temperature for the specified amount of time. Time can be passed in as minutes or seconds. In the example below, the Thermocycler Module will hold the the specified temperature for 45 minutes and 15 seconds. If you do not specify a hold time the protocol will proceed once the temperature specified is reached.

```
tc_mod.set_block_temperature(4, hold_time_seconds=15, hold_time_minutes=45)
```

Ramp Rate

Lastly, you can modify the ramp rate in degC/sec for a given temperature.

```
tc_mod.set_block_temperature(4, hold_time_seconds=60, ramp_rate=0.5)
```

Warning: Do not change this parameter unless you know what you're doing.

Thermocycler Module Profiles

Unlike the temperature module, the Thermocycler Module can rapidly cycle through temperatures to accomplish heat-sensitive reactions. To set up a Thermocycler Module profile, like you might on the UI of other Thermocycler Modules, use *ThermocyclerContext.execute_profile()* (page 117). A profile requires one or more steps, each of which must contain a temperature and a hold time. As with the *ThermocyclerContext.set_block_temperature()* (page 118) method, you have the option of specifying your hold time in seconds or minutes with `hold_time_seconds` and `hold_time_minutes`. **Note** This is *only* for controlling the temperature of the *block* in the Thermocycler Module.

```
profile = [
  {temperature: 10, hold_time_seconds: 30},
  {temperature: 10, hold_time_seconds: 30},
  {temperature: 10, hold_time_seconds: 30}]

tc_mod.execute_profile(steps=profile, repetitions=30)
```

Thermocycler Module Status

Throughout your protocol, you may want particular information on the current status of your Thermocycler Module. Below are a few methods that allow you to do that.

Lid Position

Returns the current status of the lid position. It can be one of the strings `'open'`, `'closed'` or `'in_between'`.

```
tc_mod.lid_position
```

Heated Lid Temperature Status

Returns the current status of the heated lid temperature. It can be one of the strings `'holding at target'` or `'idle'`.

```
tc_mod.lid_temperature_status
```

Block Temperature Status

Returns the current status of the well block temperature controller. It can be one of the strings `'holding at target'`, `'cooling'`, or `'heating'`.

```
tc_mod.block_temperature_status
```

Thermocycler Module Deactivate

At some points in your protocol, you may want to deactivate certain aspects of your Thermocycler Module. This can be done with three methods, *ThermocyclerContext.deactivate()* (page 117), *ThermocyclerContext.deactivate_lid()* (page 117), *ThermocyclerContext.deactivate_block()* (page 117) that will be able to turn off certain functionalities of the Thermocycler Module.

Deactivate

This deactivates both the well block and the heated lid of the Thermocycler Module.

```
tc_mod.deactivate()
```

Deactivate Lid

This deactivates only the heated lid of the Thermocycler Module.

```
tc_mod.deactivate_lid()
```

Deactivate Block

This deactivates only the well block of the Thermocycler Module.

```
tc_mod.deactivate_block()
```

Building Block Commands

Basic commands are the smallest individual actions that can be completed on a robot. For example, a liquid transfer at its core, found in *Complex Commands* (page 82), can be separated into a series of `pick_up_tip()`, `aspirate()`, `dispense()`, `drop_tip()` etc.

For the purposes of this section we can assume that we already have the following:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware('corning_96_wellplate_360ul_flat', 2)
    plate = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    pipette = protocol.load_instrument('p300_single', mount='left')
    # the example code below would go here, inside the run function
```

This loads a Corning 96 Well Plate[32] in slot 2 and a Opentrons 300ul Tiprack[33] in slot 3, and uses a P300 Single pipette.

[32] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[33] https://labware.opentrons.com/opentrons_96_tiprack_300ul

We save the result of *ProtocolContext.load_instrument()* (page 96) in the variable `pipette`. This is always an instance of the *opentrons.protocol_api.contexts.InstrumentContext* (page 99) class. Any method or attribute that has to do with a given pipette (as opposed to the OT-2 as a whole) is part of this class.

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.

Pick Up Tip

Before any liquid handling can be done, your pipette must have a tip on it. The command *InstrumentContext.pick_up_tip()* (page 105) 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['A1'])
```

If you have associated a tiprack with your pipette such as in the *Pipettes* (page 60) section or *Protocols and Instruments* (page 95), then you can simply call

```
pipette.pick_up_tip()
```

Drop Tip

Once finished with a tip, the pipette will remove the tip when we call *InstrumentContext.drop_tip()* (page 102). 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.

```
pipette.drop_tip(tiprack['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 = protocol.load_labware('trash-box', 4)
pipette.pick_up_tip()
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 *InstrumentContext.return_tip()* (page 105). The example below will automatically return the tip to `'A3'` on the tip rack.

```
pipette.pick_up_tip(tiprack['A3'])
pipette.return_tip()
```

For the purposes of this section we can assume that we already have the following:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware(
        'corning_96_wellplate_360ul_flat', 2)
    plate = protocol.load_labware(
        'opentrons_96_tiprack_300ul', 3)
    pipette = protocol.load_instrument(
        'p300_single', mount='left', tip_racks=[tiprack])
```

This loads a Corning 96 Well Plate[34] in slot 2 and a Opentrons 300ul Tiprack[35] in slot 3, and uses a P300 Single pipette.

Iterating Through Tips

Now that we have two tip racks attached to the pipette, we can automatically step through each tip whenever we call *InstrumentContext.pick_up_tip()* (page 105). We then have the option to either *InstrumentContext.return_tip()* (page 105) to the tip rack, or we can *InstrumentContext.drop_tip()* (page 102) 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 *InstrumentContext.pick_up_tip()* (page 105) 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()
```

Liquid Control

This is the fun section, where we get to move things around and pipette! This section describes the *InstrumentContext* (page 99) 's many liquid-handling commands, as well as how to command the OT-2 to a specific point. Please note that the default now for pipette aspirate and dispense location is a 1mm offset from the **bottom** of the well now.

---

[34] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[35] https://labware.opentrons.com/opentrons_96_tiprack_300ul

```
def run(protocol):
    tiprack = protocol.load_labware('corning_96_wellplate_360ul_flat', 2)
    plate = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    pipette = protocol.load_instrument('p300_single', mount='left', tip_
↪racks=[tiprack])
    pipette.pick_up_tip()
```

This loads a Corning 96 Well Plate[36] in slot 2 and a Opentrons 300ul Tiprack[37] in slot 3, and uses a P300 Single pipette.

Aspirate

To aspirate is to pull liquid up into the pipette's tip. When calling *InstrumentContext.aspirate()* (page 100) on a pipette, we can specify how many microliters, and at which location, to draw liquid from:

```
pipette.aspirate(50, plate['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 its current location (which we previously set as `plate['A1']`).

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

Now our pipette's tip is holding 100uL.

Note: By default, the OT-2 will move to 1mm above the bottom of the target well before aspirating. You can change this by using a well position function like *Well.bottom()* (page 111) (see *Specifying Position Within Wells* (page 67)) every time you call `aspirate`, or - if you want to change the default throughout your protocol - you can change the default offset with *InstrumentContext.well_bottom_clearance* (page 107) (see *Default Positions Within Wells* (page 62)).

Dispense

To dispense is to push out liquid from the pipette's tip. The usage of *InstrumentContext.dispense()* (page 102) in the Opentrons API is nearly identical to *InstrumentContext.aspirate()* (page 100), in that you can specify microliters and location, or only microliters.

```
pipette.dispense(50, plate['B1']) # dispense 50uL to plate:B1
pipette.dispense(50)              # dispense 50uL to current position
```

Note: By default, the OT-2 will move to 1mm above the bottom of the target well before dispensing. You can change this by using a well position function like *Well.bottom()* (page 111) (see *Specifying Position Within Wells* (page 67)) every time you call `dispense`, or - if you want to change the default throughout your protocol - you can change the default offset with *InstrumentContext.well_bottom_clearance* (page 107) (see *Default Positions Within Wells* (page 62)).

---

[36] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[37] https://labware.opentrons.com/opentrons_96_tiprack_300ul

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 *InstrumentContext.blow_out()* (page 101), 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 its current position.

```
pipette.blow_out()             # blow out in current location
pipette.blow_out(plate['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 *InstrumentContext.touch_tip()* (page 106) on a pipette, we have the option to specify a location where the tip will touch the inner walls.

Touch tip can take up to 4 arguments: `touch_tip(location,radius,v_offset,speed)`.

```
pipette.touch_tip()             # touch tip within current location
pipette.touch_tip(v_offset=-2) # touch tip 2mm below the top of the current location
pipette.touch_tip(plate['B1']) # touch tip within plate:B1
pipette.touch_tip(plate['B1'], # touch tip in plate:B1, at 75% of total radius and -
↪2mm from top of well
                  radius=0.75,
                  v_offset=-2)
```

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 *InstrumentContext.mix()* (page 104).

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

```
pipette.mix(4, 100, plate.['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 *InstrumentContext.air_gap()* (page 100) with a microliter amount will aspirate that much air into the tip.

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

Moving

Move To

You can use *InstrumentContext.move_to()* (page 104) to move a pipette any location on the deck.

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

```
pipette.move_to(tiprack['A1'])
```

You can also specify at what height you would like the robot to move to inside of a location using *Well.top()* (page 111) and *Well.bottom()* (page 111) methods on that location (more on these methods and others like them in the *Specifying Position Within Wells* (page 67) section):

```
pipette.move_to(plate['A1'].bottom())   # move to the bottom of well A1
pipette.move_to(plate['A1'].top())      # move to the top of well A1
pipette.move_to(plate['A1'].bottom(2)) # move to 2mm above the bottom of well A1
pipette.move_to(plate['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 move in a straight line to the target location, you can set the movement strategy to `'direct'`.

```
pipette.move_to(plate['A1'], force_direct=True)
```

Warning: Moving without an arc will run the risk of colliding with things on your deck. Be very careful when using this option.

Usually the above 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['A1'])
pipette.move_to(plate['A1'].bottom(1), force_direct=True)
pipette.move_to(plate['A1'].top(-2), force_direct=True)
pipette.move_to(plate['A2'])
```

Utility Commands

Delay

ProtocolContext.delay() (page 96) (a method of `ProtocolContext` since it concerns the robot as a whole) pauses your protocol for any given number of minutes or seconds. The value passed into `delay()` is the number of minutes or seconds the robot will wait until moving on to the next commands.

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

User-Specified Pause

The method *ProtocolContext.pause()* (page 99) will pause protocol execution at a specific step. You can resume by pressing 'resume' in your OT App. You can optionally specify a message that will be displayed in the Opentrons app when protocol execution pauses.

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    # The start of your protocol goes here...

    # The robot stops here until you press resume. It will display the message in
    # the Opentrons app. You do not need to specify a message, but it makes things
    # more clear.
    protocol.pause('Time to take a break')
```

Homing

You can manually request that the robot home during protocol execution. This is typically not necessary; however, if you have some custom labware or setup that you suspect may make the robot crash or skip steps, or if at any point you will disengage motors or move the gantry with your hand, you may want to command a home afterwards.

To home the entire robot, you can call *ProtocolContext.home()* (page 96).

To home a specific pipette's Z stage and plunger, you can call *InstrumentContext.home()* (page 103).

To home a specific pipette's plunger only, you can call *InstrumentContext.home_plunger()* (page 103).

None of these functions take any arguments:

```
from opentrons import protocol_api, types

def run(protocol: protocol_api.ProtocolContext):
    pipette = protocol.load_instrument('p300_single', 'right')
    protocol.home() # Homes the gantry, z axes, and plungers
    pipette.home()  # Homes the right z axis and plunger
    pipette.home_plunger() # Homes the right plunger
```

Comment

The method *ProtocolContext.comment()* (page 96) lets you display messages in the Opentrons app during protocol execution:

```
from opentrons import protocol_api, types

def run(protocol: protocol_api.ProtocolContext):
    protocol.comment('Hello, world!')
```

Complex Commands

Overview

The difference between this section and *Building Block Commands* (page 76) is ease of use. With complex liquid handling commands, you can more easily handle larger groups of wells to perform repetitive actions. The main

downside to using complex liquid handling commands is that you cannot control the order in which operations are executed. In this section we will order in which operations are executed. In this section we explain when and how specific actions are executed.

The examples in this section will use the following set-up:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    pipette = protocol.load_instrument('p300_single', mount='left', tip_
↪racks=[tiprack])

    # The code used in the rest of the examples goes here
```

This loads a Corning 96 Well Plate[38] in slot 1 and a Opentrons 300ul Tiprack[39] in slot 2, and uses a P300 Single pipette.

You can follow along and simulate the protocol using our protocol simulator, which can be installed by following the instructions at *Design with Python* (page 3).

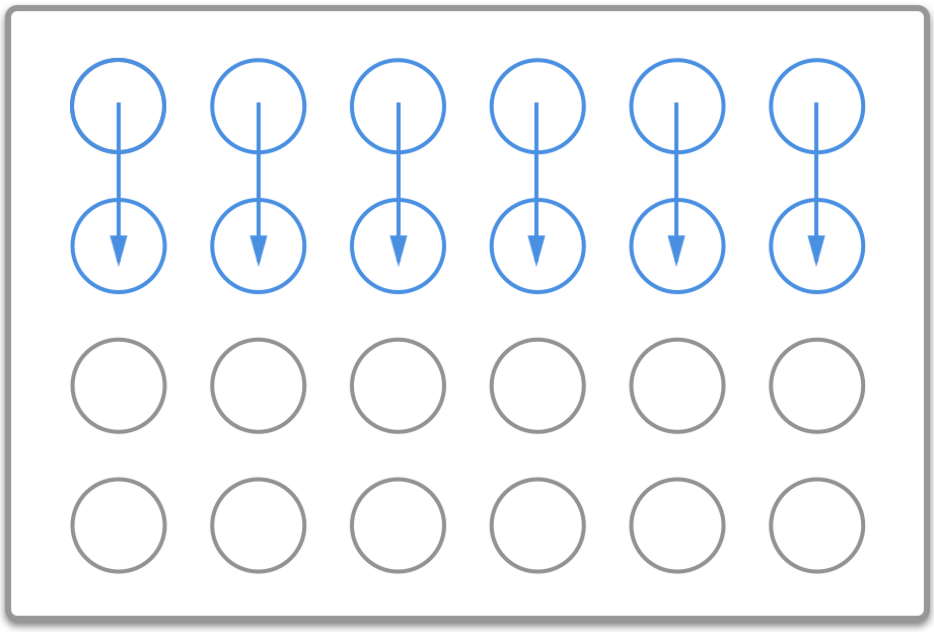
There are three general complex liquid handling commands. The differences can be found in the table below

Method	One source well to a group of destination wells	Many source wells to a group of destination wells	Many source wells to one destination well
InstrumentContext.transfer() (page 106)	Yes	Yes	Yes
InstrumentContext.distribute() (page 102)	Yes	No	No
InstrumentContext.consolidate() (page 101)	No	No	No

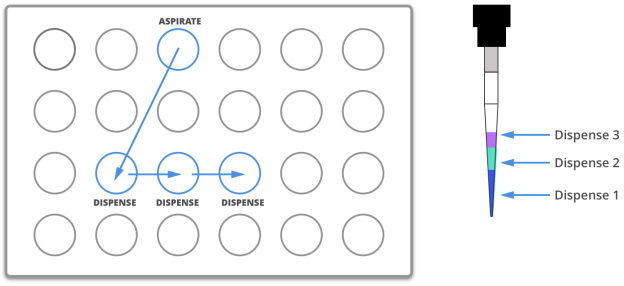
You can also refer to these images for further clarification.

[38] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[39] https://labware.opentrons.com/opentrons_96_tiprack_300ul

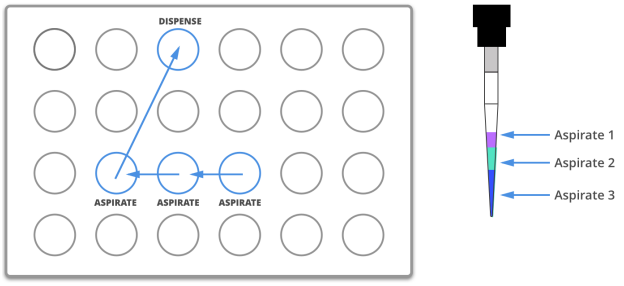
Transfer



Distribute



Consolidate



Parameters

Parameters for our complex liquid handling listed in order of operation. Check out the *Complex Liquid Handling Parameters* (page 91) section for examples on how to use these parameters.

Parameter(s)	Options	Transfer Defaults	Distribute Defaults	Consolidate Defaults
`new_tip`	`'always'`, `'never'`, `'once'`	`'once'`	`'once'`	`'once'`
`mix_before`, `mix_after`	`mix_before` and `mix_after` require a tuple of (repetitions, volume)	No mixing either before aspirate or after dispense	No mixing before aspirate, mixing after dispense is ignored	Mixing before aspirate is ignored, no mix after dispense by default
`touch_tip`	`True` or `False`, if true touch tip on both source and destination wells	No touch tip by default	No touch tip by default	No touch tip by default
`air_gap`	Volume in microliters	0	0	0
`blow_out`	`True` or `False`, if true blow out at dispense	`False`	`False`	`False`
`trash`	`True` or `False`, if false return tip to tiprack	`True`	`True`	`True`
`carryover`	`True` or `False`, if true split volumes that exceed max volume of pipette into smaller quantities	`True`	`False`	`False`
`disposal_volume`	Extra volume in microliters to hold in tip while dispensing; better accuracies in multi-dispense	0	10% of pipette max volume	0

Transfer

The most versatile of the complex liquid handling functions is *`InstrumentContext.transfer()`* (page 106). For a majority of use-cases you will most likely want to use this complex command. Below you will find a few scenarios utilizing the *`InstrumentContext.transfer()`* (page 106) command.

Basic

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

```
pipette.transfer(100, plate.wells_by_name()['A1'], plate.wells_by_name()['B1'])
```

Transfer commands will automatically create entire series of *`InstrumentContext.aspirate()`* (page 100), *`InstrumentContext.dispense()`* (page 102), and other `InstrumentContext()` commands.

Large Volumes

Volumes larger than the pipette's `max_volume` *Defaults* (page 63) will automatically divide into smaller transfers.

```
pipette.transfer(700, plate.wells_by_name()['A2'], plate.wells_by_name()['B2'])
```

will have the steps...

```
Transferring 700 from well A2 in "1" to well B2 in "1"
Picking up tip well A1 in "2"
Aspirating 300.0 uL from well A2 in "1" at 1 speed
Dispensing 300.0 uL into well B2 in "1"
Aspirating 200.0 uL from well A2 in "1" at 1 speed
Dispensing 200.0 uL into well B2 in "1"
Aspirating 200.0 uL from well A2 in "1" at 1 speed
Dispensing 200.0 uL into well B2 in "1"
Dropping tip well A1 in "12"
```

One to One

Transfer commands are most useful when moving liquid between multiple wells. Notice this will be a one to one transfer from where well `A1`'s contents are transferred to well `A2`, and so on and so forth. Refer to transfer for better visualization.

```
pipette.transfer(100, plate.columns_by_name()['1'], plate.columns_by_name()['2'])
```

will have the steps...

```
Transferring 100 from wells A1...H1 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Aspirating 100.0 uL from well B1 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well C1 in "1" at 1 speed
Dispensing 100.0 uL into well C2 in "1"
Aspirating 100.0 uL from well D1 in "1" at 1 speed
Dispensing 100.0 uL into well D2 in "1"
Aspirating 100.0 uL from well E1 in "1" at 1 speed
Dispensing 100.0 uL into well E2 in "1"
Aspirating 100.0 uL from well F1 in "1" at 1 speed
Dispensing 100.0 uL into well F2 in "1"
Aspirating 100.0 uL from well G1 in "1" at 1 speed
Dispensing 100.0 uL into well G2 in "1"
Aspirating 100.0 uL from well H1 in "1" at 1 speed
Dispensing 100.0 uL into well H2 in "1"
Dropping tip well A1 in "12"
```

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_by_name()['A1'], plate.columns_by_name()['2'])
```

will have the steps...

```
Transferring 100 from well A1 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well C2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well D2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well E2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well F2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well G2 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well H2 in "1"
Dropping tip well A1 in "12"
```

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['A1'],
    [plate.wells_by_name()[well_name] for well_name in ['B1', 'B2', 'B3']])
```

will have the steps...

```
Transferring [20, 40, 60] from well A1 in "1" to wells B1...B3 in "1"
Picking up tip well A1 in "2"
Aspirating 20.0 uL from well A1 in "1" at 1 speed
Dispensing 20.0 uL into well B1 in "1"
Aspirating 40.0 uL from well A1 in "1" at 1 speed
Dispensing 40.0 uL into well B2 in "1"
Aspirating 60.0 uL from well A1 in "1" at 1 speed
Dispensing 60.0 uL into well B3 in "1"
Dropping tip well A1 in "12"
```

Distribute and Consolidate

Save time and tips with the *InstrumentContext.distribute()* (page 102) and *InstrumentContext.consolidate()* (page 101) commands. These are nearly identical to *InstrumentContext.transfer()* (page 106), except that they will combine multiple transfers 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. Refer to consolidate for better visualization.

```
pipette.consolidate(30, plate.columns_by_name()['2'], plate.wells_by_name()['A1'])
```

will have the steps...

```
Consolidating 30 from wells A2...H2 in "1" to well A1 in "1"
Transferring 30 from wells A2...H2 in "1" to well A1 in "1"
Picking up tip well A1 in "2"
Aspirating 30.0 uL from well A2 in "1" at 1 speed
Aspirating 30.0 uL from well B2 in "1" at 1 speed
Aspirating 30.0 uL from well C2 in "1" at 1 speed
Aspirating 30.0 uL from well D2 in "1" at 1 speed
Aspirating 30.0 uL from well E2 in "1" at 1 speed
Aspirating 30.0 uL from well F2 in "1" at 1 speed
Aspirating 30.0 uL from well G2 in "1" at 1 speed
Aspirating 30.0 uL from well H2 in "1" at 1 speed
Dispensing 240.0 uL into well A1 in "1"
Dropping tip well A1 in "12"
```

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

```
pipette.consolidate(
  30,
  plate.columns_by_name()['1'],
  [plate.wells_by_name()[well_name] for well_name in ['A1', 'A2']])
```

will have the steps...

```
Consolidating 30 from wells A1...H1 in "1" to wells A1...A2 in "1"
Transferring 30 from wells A1...H1 in "1" to wells A1...A2 in "1"
Picking up tip well A1 in "2"
Aspirating 30.0 uL from well A1 in "1" at 1 speed
Aspirating 30.0 uL from well B1 in "1" at 1 speed
Aspirating 30.0 uL from well C1 in "1" at 1 speed
Aspirating 30.0 uL from well D1 in "1" at 1 speed
Dispensing 120.0 uL into well A1 in "1"
Aspirating 30.0 uL from well E1 in "1" at 1 speed
Aspirating 30.0 uL from well F1 in "1" at 1 speed
Aspirating 30.0 uL from well G1 in "1" at 1 speed
Aspirating 30.0 uL from well H1 in "1" at 1 speed
Dispensing 120.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

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_by_name()['A1'], plate.rows_by_name()['A'])
```

will have the steps...

```
Distributing 55 from well A1 in "1" to wells A1...A12 in "1"
Transferring 55 from well A1 in "1" to wells A1...A12 in "1"
Picking up tip well A1 in "2"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A1 in "1"
Dispensing 55.0 uL into well A2 in "1"
Dispensing 55.0 uL into well A3 in "1"
Dispensing 55.0 uL into well A4 in "1"
Blowing out at well A1 in "12"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A5 in "1"
Dispensing 55.0 uL into well A6 in "1"
Dispensing 55.0 uL into well A7 in "1"
Dispensing 55.0 uL into well A8 in "1"
Blowing out at well A1 in "12"
Aspirating 250.0 uL from well A1 in "1" at 1 speed
Dispensing 55.0 uL into well A9 in "1"
Dispensing 55.0 uL into well A10 in "1"
Dispensing 55.0 uL into well A11 in "1"
Dispensing 55.0 uL into well A12 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

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

```
pipette.distribute(
  30,
  [plate.wells_by_name()[well_name] for well_name in ['A1', 'A2']],
  plate.rows()['A'])
```

will have the steps...

```
Distributing 30 from wells A1...A2 in "1" to wells A1...A12 in "1"
Transferring 30 from wells A1...A2 in "1" to wells A1...A12 in "1"
Picking up tip well A1 in "2"
Aspirating 210.0 uL from well A1 in "1" at 1 speed
Dispensing 30.0 uL into well A1 in "1"
Dispensing 30.0 uL into well A2 in "1"
Dispensing 30.0 uL into well A3 in "1"
Dispensing 30.0 uL into well A4 in "1"
Dispensing 30.0 uL into well A5 in "1"
Dispensing 30.0 uL into well A6 in "1"
Blowing out at well A1 in "12"
Aspirating 210.0 uL from well A2 in "1" at 1 speed
Dispensing 30.0 uL into well A7 in "1"
Dispensing 30.0 uL into well A8 in "1"
Dispensing 30.0 uL into well A9 in "1"
Dispensing 30.0 uL into well A10 in "1"
Dispensing 30.0 uL into well A11 in "1"
Dispensing 30.0 uL into well A12 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

Re-Visiting Order of Operations

Given this sample code, what is the order of operations?

```
pipette.transfer(
    100,
    [plate.wells_by_name()[well_name] for well_name in ['A1', 'A2', 'A3']],
    [plate.wells_by_name()[well_name] for well_name in ['B1', 'B2', 'B3']],
    new_tip='always',
    disposal_volume=10,
    touch_tip=True,
    air_gap=10,
    mix_before=(2, 50),
    mix_after=(2, 50),
    blow_out=True)
```

The order in which the parameters are listed inside of a complex method are irrelevant. Instead, the order in which parameters are executed is as follows:

1. Tip logic
2. Mix at source location
3. Aspirate + Any disposal volume
4. Touch tip
5. Air gap
6. Dispense
7. Touch tip

<——Repeat above for all wells——>

8. Empty disposal volume into trash
9. Blow Out

Notice how blow out only occurs after getting rid of disposal volume. If you want blow out to occur after every dispense, you should not include a disposal volume.

Which Command Should I Use?

Now that you know a little more about the different complex liquid handling options, which one should you use?

Each method handles groups of wells differently. We tried to encapsulate the different options you might encounter when utilizing complex commands in the table below.

Method	One source well to a group of destination wells	Many source wells to a group of destination wells	Many source wells to one destination well
InstrumentContext.transfer() (page 106)	Yes	Yes	Yes
InstrumentContext.distribute() (page 102)	Yes	No	No
InstrumentContext.consolidate() (page 101)	No	No	No

You can also check out this other table below on how each method compares for things such as contamination or speed. If a method is intended for a particular category, it is marked with an X.

	Accuracy	Speed	Waste Reduction	Contamination
Transfer	X			X
Distribute		X	X	
Consolidate		X	X	

Complex Liquid Handling Parameters

Below are some examples of the parameters described in *Parameters* (page 85).

new_tip

This parameter handles tip logic. You have options of `always`, `once` and `never`. The default for every complex command is `once`.

If you want to avoid cross-contamination and increase accuracy, you should set this parameter to `always`.

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_by_name()[well_name] for well_name in ['A1', 'A2', 'A3']],
    [plate.wells_by_name()[well_name] for well_name in ['B1', 'B2', 'B3']],
    new_tip='always')     # always pick up a new tip
```

will have the steps...

```
Transferring 100 from wells A1...A3 in "1" to wells B1...B3 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Dropping tip well A1 in "12"
Picking up tip well B1 in "2"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Dropping tip well A1 in "12"
Picking up tip well C1 in "2"
Aspirating 100.0 uL from well A3 in "1" at 1 speed
Dispensing 100.0 uL into well B3 in "1"
Dropping tip well A1 in "12"
```

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.pick_up_tip()
...
pipette.transfer(
    100,
    [plate.wells_by_name()[well_name] for well_name in ['A1', 'A2', 'A3']],
    [plate.wells_by_name()[well_name] for well_name in ['B1', 'B2', 'B3']],
    new_tip='never')     # never pick up or drop a tip
...
pipette.drop_tip()
```

will have the steps...

```
Picking up tip well A1 in "2"
...
Transferring 100 from wells A1...A3 in "1" to wells B1...B3 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Aspirating 100.0 uL from well A2 in "1" at 1 speed
Dispensing 100.0 uL into well B2 in "1"
Aspirating 100.0 uL from well A3 in "1" at 1 speed
Dispensing 100.0 uL into well B3 in "1"
...
Dropping tip well A1 in "12"
```

trash

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 its tip rack, you can set `trash=False`.

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

will have the steps...

```
Transferring 100 from well A1 in "1" to well B1 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well B1 in "1"
Returning tip
Dropping tip well A1 in "2"
```

touch_tip

A *Touch Tip* (page 80) can be performed after every aspirate and dispense by setting `touch_tip=True`.

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

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Touching tip
Dispensing 100.0 uL into well A2 in "1"
Touching tip
Dropping tip well A1 in "12"
```

blow_out

A *Blow Out* (page 80) can be performed after every dispense that leaves the tip empty by setting `blow_out=True`.

```
pipette.transfer(
    100,
    plate['A1'],
    plate['A2'],
    blow_out=True)       # blow out droplets when tip is empty
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Blowing out
Dropping tip well A1 in "12"
```

mix_before, mix_after

A *Mix* (page 80) 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['A1'],
    plate['A2'],
    mix_before=(2, 50), # mix 2 times with 50uL before aspirating
    mix_after=(3, 75))  # mix 3 times with 75uL after dispensing
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Mixing 2 times with a volume of 50ul
Aspirating 50 uL from well A1 in "1" at 1.0 speed
Dispensing 50 uL into well A1 in "1"
Aspirating 50 uL from well A1 in "1" at 1.0 speed
Dispensing 50 uL into well A1 in "1"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Dispensing 100.0 uL into well A2 in "1"
Mixing 3 times with a volume of 75ul
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
```

```
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
Aspirating 75 uL from well A2 in "1" at 1.0 speed
Dispensing 75.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

air_gap

An *Air Gap* (page 80) 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['A1'],
    plate['A2'],
    air_gap=20)          # add 20uL of air after each aspirate
```

will have the steps...

```
Transferring 100 from well A1 in "1" to well A2 in "1"
Picking up tip well A1 in "2"
Aspirating 100.0 uL from well A1 in "1" at 1 speed
Air gap
Aspirating 20 uL from well A1 in "1" at 1.0 speed
Dispensing 20 uL into well A2 in "1"
Dispensing 100.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

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.

There is a default disposal volume (equal to the pipette's minimum volume *Defaults* (page 63)), which will be blown out at the trash after the dispenses.

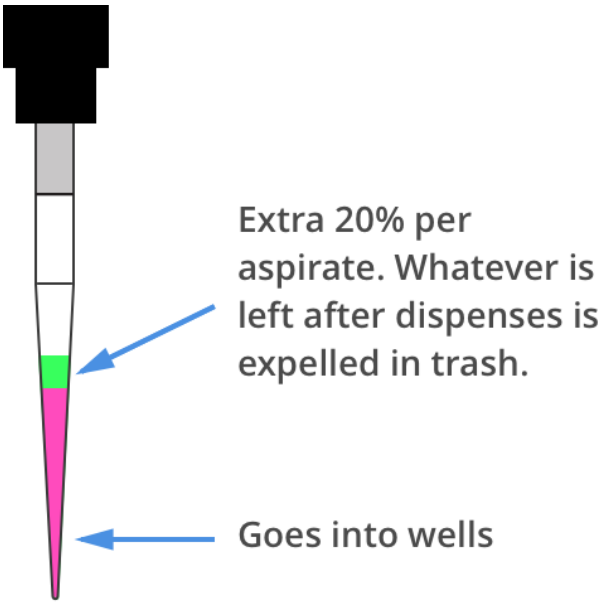
```
pipette.distribute(
    30,
    [plate.wells_by_name()[well_name] for well_name in ['A1', 'A2']],
    plate.columns_by_name()['2'],
    disposal_volume=60)    # include extra liquid to make dispenses more accurate, 20%␣
↪of total volume
```

will have the steps...

```
Distributing 30 from wells A1...A2 in "1" to wells A2...H2 in "1"
Transferring 30 from wells A1...A2 in "1" to wells A2...H2 in "1"
Picking up tip well A1 in "2"
Aspirating 130.0 uL from well A1 in "1" at 1 speed
Dispensing 30.0 uL into well A2 in "1"
Dispensing 30.0 uL into well B2 in "1"
Dispensing 30.0 uL into well C2 in "1"
Dispensing 30.0 uL into well D2 in "1"
Blowing out at well A1 in "12"
```

```
Aspirating 130.0 uL from well A2 in "1" at 1 speed
Dispensing 30.0 uL into well E2 in "1"
Dispensing 30.0 uL into well F2 in "1"
Dispensing 30.0 uL into well G2 in "1"
Dispensing 30.0 uL into well H2 in "1"
Blowing out at well A1 in "12"
Dropping tip well A1 in "12"
```

See this image for example,



API Version 2 Reference

Protocols and Instruments

class opentrons.protocol_api.contexts.**ProtocolContext**(*loop: asyncio.events.AbstractEventLoop = None*, *hardware: opentrons.hardware_control.API = None*, *broker=None*, *bundled_labware: typing.Dict[str, typing.Dict[str, typing.Any]] = None*, *bundled_data: typing.Dict[str, bytes] = None*, *extra_labware: typing.Dict[str, typing.Dict[str, typing.Any]] = None*) → None

The Context class is a container for the state of a protocol.

It encapsulates many of the methods formerly found in the Robot class, including labware, instrument, and module loading, as well as core functions like pause and resume.

Unlike the old robot class, it is designed to be ephemeral. The lifetime of a particular instance should be about the same as the lifetime of a protocol. The only exception is the one stored in `back_compat.robot`, which is provided only for back compatibility and should be used less and less as time goes by.

bundled_data
Accessor for data files bundled with this protocol, if any.

This is a dictionary mapping the filenames of bundled datafiles, with extensions but without paths (e.g. if a file is stored in the bundle as `data/mydata/aspirations.csv` it will be in the dict as `'aspirations.csv'`) to the bytes contents of the files.

comment(*msg*)
Add a user-readable comment string that will be echoed to the Opentrons app.

config
Get the robot's configuration object.

Returns .robot_config The loaded configuration.

connect(*hardware: opentrons.hardware_control.API*)
Connect to a running hardware API.

This can be either a simulator or a full hardware controller.

Note that there is no true disconnected state for a *ProtocolContext* (page 95); *disconnect()* (page 96) simply creates a new simulator and replaces the current hardware with it.

deck
The object holding the deck layout of the robot.

This object behaves like a dictionary with keys for both numeric and string slot numbers (for instance, `protocol.deck[1]` and `protocol.deck['1']` will both return the object in slot 1). If nothing is loaded into a slot, `None` will be present. This object is useful for determining if a slot in the deck is free. Rather than filtering the objects in the deck map yourself, you can also use *loaded_labwares* (page 98) to see a dict of labwares and *loaded_modules* (page 98) to see a dict of modules.

delay(*seconds=0*, *minutes=0*, *msg=None*)
Delay protocol execution for a specific amount of time.

Parameters

- **seconds** (*float*) – A time to delay in seconds
- **minutes** (*float*) – A time to delay in minutes

If both *seconds* and *minutes* are specified, they will be added.

disconnect()
Disconnect from currently-connected hardware and simulate instead

fixed_trash
The trash fixed to slot 12 of the robot deck.

home()
Homes the robot.

load_instrument(*instrument_name: str, mount: typing.Union[opentrons.types.Mount, str], tip_racks: typing.List[opentrons.protocol_api.labware.Labware] = None, replace: bool = False*) → opentrons.protocol_api.contexts.InstrumentContext
Load a specific instrument required by the protocol.

This value will actually be checked when the protocol runs, to ensure that the correct instrument is attached in the specified location.

Parameters

- **instrument_name** (*str*) – The name of the instrument model, or a prefix. For instance, 'p10_single' may be used to request a P10 single regardless of the version.

- **mount** (*types.Mount or str*) – The mount in which this instrument should be attached. This can either be an instance of the enum type *types.Mount* (page 119) or one of the strings *'left'* and *'right'*.
- **tip_racks** (List[*Labware* (page 108)]) – A list of tip racks from which to pick tips if *InstrumentContext.pick_up_tip()* (page 105) is called without arguments.
- **replace** (*bool*) – Indicate that the currently-loaded instrument in *mount* (if such an instrument exists) should be replaced by *instrument_name*.

load_labware (*load_name: str, location: typing.Union[int, str], label: str = None, namespace: str = None, version: int = None*) → opentrons.protocol_api.labware.Labware

Load a labware onto the deck given its name.

For labware already defined by Opentrons, this is a convenient way to collapse the two stages of labware initialization (creating the labware and adding it to the protocol) into one.

This function returns the created and initialized labware for use later in the protocol.

Parameters

- **load_name** – A string to use for looking up a labware definition
- **location** (*int or str*) – The slot into which to load the labware such as 1 or '1'
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.

load_labware_from_definition (*labware_def: typing.Dict[str, typing.Any], location: typing.Union[int, str], label: str = None*) → opentrons.protocol_api.labware.Labware

Specify the presence of a piece of labware on the OT2 deck.

This function loads the labware definition specified by *labware_def* to the location specified by *location*.

Parameters

- **labware_def** – The labware definition to load
- **location** (*int or str*) – The slot into which to load the labware such as 1 or '1'

load_module (*module_name: str, location: typing.Union[int, str, NoneType] = None*) → typing.Union[_ForwardRef('TemperatureModuleContext'), _ForwardRef('MagneticModuleContext'), _ForwardRef('ThermocyclerContext')]

Load a module onto the deck given its name.

This is the function to call to use a module in your protocol, like *load_instrument()* (page 96) is the method to call to use an instrument in your protocol. It returns the created and initialized module context, which will be a different class depending on the kind of module loaded.

A map of deck positions to loaded modules can be accessed later using *loaded_modules* (page 98).

Parameters

- **module_name** (*str*) – The name of the module.
- **location** (*str or int or None*) – The location of the module. This is usually the name or number of the slot on the deck where you will be placing the module. Some

modules, like the Thermocycler, are only valid in one deck location. You do not have to specify a location when loading a Thermocycler - it will always be in Slot 7.

Returns ModuleContext The loaded and initialized `ModuleContext`.

`loaded_instruments`

Get the instruments that have been loaded into the protocol.

This is a map of mount name to instruments previously loaded with *`load_instrument()`* (page 96). It is not necessarily the same as the instruments attached to the robot - for instance, if the robot has an instrument in both mounts but your protocol has only loaded one of them with *`load_instrument()`* (page 96), the unused one will not be present.

Returns A dict mapping mount names in lowercase to the instrument in that mount. If no instrument is loaded in the mount, it will not be present

`loaded_labwares`

Get the labwares that have been loaded into the protocol context.

Slots with nothing in them will not be present in the return value.

Note: If a module is present on the deck but no labware has been loaded into it with `ModuleContext.load_labware()`, there will be no entry for that slot in this value. That means you should not use `loaded_labwares` to determine if a slot is available or not, only to get a list of labwares. If you want a data structure of all objects on the deck regardless of type, see *`deck`* (page 96).

Returns Dict mapping deck slot number to labware, sorted in order of the locations.

`loaded_modules`

Get the modules loaded into the protocol context.

This is a map of deck positions to modules loaded by previous calls to *`load_module()`* (page 97). It is not necessarily the same as the modules attached to the robot - for instance, if the robot has a Magnetic Module and a Temperature Module attached, but the protocol has only loaded the Temperature Module with *`load_module()`* (page 97), only the Temperature Module will be present.

Returns Dict[str, ModuleContext] Dict mapping slot name to module contexts. The elements may not be ordered by slot number.

`location_cache`

The cache used by the robot to determine where it last was.

`max_speeds`

Per-axis speed limits when moving this instrument.

Changing this value changes the speed limit for each non-plunger axis of the robot, when moving this pipette. Note that this does only sets a limit on how fast movements can be; movements can still be slower than this. However, it is useful if you require the robot to move much more slowly than normal when using this pipette.

This is a dictionary mapping string names of axes to float values limiting speeds. To change a speed, set that axis's value. To reset an axis's speed to default, delete the entry for that axis or assign it to `None`.

For instance,

```
def run(protocol):
    protocol.comment(str(right.max_speeds))  # '{}' - all default
    protocol.max_speeds['A'] = 10  # limit max speed of
                                   # right pipette Z to 10mm/s
```

```
    del protocol.max_speeds['A']  # reset to default
    protocol.max_speeds['X'] = 10  # limit max speed of x to
                                   # 10 mm/s
    protocol.max_speeds['X'] = None  # reset to default
```

pause(*msg=None*)

Pause execution of the protocol until resume is called.

This function returns immediately, but the next function call that is blocked by a paused robot (anything that involves moving) will not return until *resume()* (page 99) is called.

Parameters **msg** (*str*) – A message to echo back to connected clients.

reset()

Reset the state of the context and the hardware.

For instance, this call will - reset all cached knowledge about attached tips - unload all labware - unload all instruments - clear all location and instrument caches

The only state that will be kept is the position of the robot.

resume()

Resume a previously-paused protocol

set_bundle_contents(*bundled_labware: typing.Dict[str, typing.Dict[str, typing.Any]] = None, bundled_data: typing.Dict[str, bytes] = None, extra_labware: typing.Dict[str, typing.Dict[str, typing.Any]] = None*)

Specify bundle contents after the context is created. Replaces the old values.

temp_connect(*hardware: opentrons.hardware_control.API*)

Connect temporarily to the specified hardware controller.

This should be used as a context manager:

```
with ctx.temp_connect(hw):
    # do some tasks
    ctx.home()
# after the with block, the context is connected to the same
# hardware control API it was connected to before, even if
# an error occurred in the code inside the with block
```

update_config(***kwargs*)

Update values of the robot's configuration.

kwargs should contain keys of the robot's configuration. For instace, *update_config(name='Grace Hopper')* would change the name of the robot

Documentation on keys can be found in the documentation for `robot_config`.

class opentrons.protocol_api.contexts.**InstrumentContext**(*ctx: opentrons.protocol_api.contexts.ProtocolContext*, *hardware_mgr: opentrons.protocol_api.util.HardwareManager*, *mount: opentrons.types.Mount*, *log_parent: logging.Logger*, *tip_racks: typing.List[opentrons.protocol_api.labware.Labware] = None*, *trash: opentrons.protocol_api.labware.Labware = None*, *default_speed: float = 400.0*, ***config_kwargs*) → None

A context for a specific pipette or instrument.

This can be used to call methods related to pipettes - moves or aspirates or dispenses, or higher-level methods.

Instances of this class bundle up state and config changes to a pipette - for instance, changes to flow rates or trash containers. Action methods (like *aspirate()* (page 100) or *distribute()* (page 102)) are defined here for convenience.

In general, this class should not be instantiated directly; rather, instances are returned from `ProtcolContext.load_instrument()`.

air_gap(*volume: float = None*, *height: float = None*) → opentrons.protocol_api.contexts.InstrumentContext

Pull air into the pipette current tip at the current location

Parameters

- **volume** (*float*) – The amount in uL to aspirate air into the tube. (Default will use all remaining volume in tip)
- **height** (*float*) – The number of millimiters to move above the current Well to air-gap aspirate. (Default: 5mm above current Well)

Raises

- **NoTipAttachedError** – If no tip is attached to the pipette
- **RuntimeError** – If location cache is None. This should happen if *touch_tip* is called without first calling a method that takes a location (eg, *aspirate()* (page 100), *dispense()* (page 102))

Returns This instance

Note: Both `volume` and height are optional, but unlike previous API versions, if you want to specify only `height` you must do it as a keyword argument: `pipette.air_gap(height=2)`. If you call `air_gap` with only one unnamed argument, it will always be interpreted as a volume.

---

aspirate(*volume: float = None*, *location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None*, *rate: float = 1.0*) → opentrons.protocol_api.contexts.InstrumentContext

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

If only a volume is passed, the pipette will aspirate from its current position. If only a location is passed (as in `instr.aspirate(location=wellplate['A1'])`, *aspirate()* (page 100) will default to the amount of volume available.

Parameters

- **volume** (*int or float*) – The volume to aspirate, in microliters. If not specified, *max_volume* (page 103).
- **location** – Where to aspirate from. If *location* is a *Well* (page 111), the robot will aspirate from well_bottom_clearance``.aspirate`` mm above the bottom of the well. If *location* is a *Location* (page 118) (i.e. the result of *Well.top()* (page 111) or *Well.bottom()* (page 111)), the robot will aspirate from the exact specified location. If unspecified, the robot will aspirate from the current position.
- **rate** (*float*) – The relative plunger speed for this aspirate. During this aspirate, the speed of the plunger will be *rate* * aspirate_speed. If not specified, defaults to 1.0 (speed will not be modified).

Returns This instance.

Note: If aspirate is called with a single argument, it will not try to guess whether the argument is a volume or location - it is required to be a volume. If you want to call aspirate with only a location, specify it as a keyword argument: instr.aspirate(location=wellplate['A1'])

---

blow_out(*location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None*) → opentrons.protocol_api.contexts.InstrumentContext

Blow liquid out of the tip.

If *dispense* (page 102) is used to completely empty a pipette, usually a small amount of liquid will remain in the tip. This method moves the plunger past its usual stops to fully remove any remaining liquid from the tip. Regardless of how much liquid was in the tip when this function is called, after it is done the tip will be empty.

Parameters **location** (*Well* (page 111) or *Location* (page 118) or None) – The location to blow out into. If not specified, defaults to the current location of the pipette

Raises **RuntimeError** – If no location is specified and location cache is None. This should happen if *blow_out* is called without first calling a method that takes a location (eg, *aspirate()* (page 100), *dispense()* (page 102))

Returns This instance

channels

The number of channels on the pipette.

consolidate(*volume: float, source: typing.List[opentrons.protocol_api.labware.Well], dest: opentrons.protocol_api.labware.Well, *args, **kwargs*) → opentrons.protocol_api.contexts.InstrumentContext

Move liquid from multiple wells (sources) to a single well(destination)

Parameters

- **volume** – The amount of volume to consolidate from each source well.
- **source** – List of wells from where liquid will be aspirated.
- **dest** – The single well into which liquid will be dispensed.
- **kwargs** – See *transfer()* (page 106). Some arguments are changed. Specifically, - mix_before, if specified, is ignored. - disposal_volume is ignored and set to 0.

Returns This instance

current_volume

The current amount of liquid, in microliters, held in the pipette.

default_speed
The speed at which the robot's gantry moves.

By default, 400 mm/s. Changing this value will change the speed of the pipette when moving between labware. In addition to changing the default, the speed of individual motions can be changed with the `speed` argument to *`InstrumentContext.move_to()`* (page 104).

dispense(*volume: float = None, location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None, rate: float = 1.0*) → opentrons.protocol_api.contexts.InstrumentContext
Dispense a volume of liquid (in microliters/uL) using this pipette into the specified location.

If only a volume is passed, the pipette will dispense from its current position. If only a location is passed (as in `instr.dispense(location=wellplate['A1'])`), all of the liquid aspirated into the pipette will be dispensed (this volume is accessible through *`current_volume`* (page 101)).

Parameters

- **volume** (*int or float*) – The volume of liquid to dispense, in microliters. If not specified, defaults to *`current_volume`* (page 101).
- **location** – Where to dispense into. If *location* is a *`Well`* (page 111), the robot will dispense into `well_bottom_clearance``.dispense`` mm above the bottom of the well. If *location* is a *`Location`* (page 118) (i.e. the result of *`Well.top()`* (page 111) or *`Well.bottom()`* (page 111)), the robot will dispense into the exact specified location. If unspecified, the robot will dispense into the current position.
- **rate** (*float*) – The relative plunger speed for this dispense. During this dispense, the speed of the plunger will be *rate* * `dispense_speed`. If not specified, defaults to 1.0 (speed will not be modified).

Returns This instance.

Note: If `dispense` is called with a single argument, it will not try to guess whether the argument is a volume or location - it is required to be a volume. If you want to call `dispense` with only a location, specify it as a keyword argument: `instr.dispense(location=wellplate['A1'])`

---

distribute(*volume: float, source: opentrons.protocol_api.labware.Well, dest: typing.List[opentrons.protocol_api.labware.Well], *args, **kwargs*) → opentrons.protocol_api.contexts.InstrumentContext
Move a volume of liquid from one source to multiple destinations.

Parameters

- **volume** – The amount of volume to distribute to each destination well.
- **source** – A single well from where liquid will be aspirated.
- **dest** – List of Wells where liquid will be dispensed to.
- **kwargs** – See *`transfer()`* (page 106). Some arguments are changed. Specifically, - `mix_after`, if specified, is ignored. - `disposal_volume`, if not specified, is set to the

  minimum volume of the pipette

Returns This instance

drop_tip(*location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None*) → opentrons.protocol_api.contexts.InstrumentContext
Drop the current tip.

If no location is passed, the Pipette will drop the tip into its *trash_container* (page 107), which if not specified defaults to the fixed trash in slot 12.

The location in which to drop the tip can be manually specified with the *location* argument. The *location* argument can be specified in several ways:

- If the only thing to specify is which well into which to drop a tip, *location* can be a *Well* (page 111). For instance, if you have a tip rack in a variable called *tiprack*, you can drop a tip into a specific well on that tiprack with the call *instr.drop_tip(tiprack.wells()[0])*. This style of call can be used to make the robot drop a tip into arbitrary labware.
- If the position to drop the tip from as well as the *Well* (page 111) to drop the tip into needs to be specified, for instance to tell the robot to drop a tip from an unusually large height above the tiprack, *location* can be a *types.Location* (page 118); for instance, you can call *instr.drop_tip(tiprack.wells()[0].top())*.

Note: OT1 required homing the plunger after dropping tips, so the prior version of *drop_tip* automatically homed the plunger. This is no longer needed in OT2. If you need to home the plunger, use *home_plunger()* (page 103).

Parameters **location** (*types.Location* (page 118) or *Well* (page 111) or None) – The location to drop the tip

Returns This instance

flow_rate

The speeds (in uL/s) configured for the pipette.

This is an object with attributes `aspirate`, `dispense`, and `blow_out` holding the flow rates for the corresponding operation.

Note: This property is equivalent to *speed* (page 105); the only difference is the units in which this property is specified. specifiying this property uses the units of the volumetric flow rate of liquid into or out of the tip, while *speed* (page 105) uses the units of the linear speed of the plunger inside the pipette. Because *speed* (page 105) and *flow_rate* (page 103) modify the same values, setting one will override the other.

For instance, to change the flow rate for aspiration on an instrument you would do

```
instrument.flow_rate.aspirate = 50
```

home() → opentrons.protocol_api.contexts.InstrumentContext

Home the robot.

Returns This instance.

home_plunger() → opentrons.protocol_api.contexts.InstrumentContext

Home the plunger associated with this mount

Returns This instance.

hw_pipette

View the information returned by the hardware API directly.

Raises a *types.PipetteNotAttachedError* (page 119) if the pipette is no longer attached (should not happen).

max_volume
The maximum volume, in microliters, this pipette can hold.

mix(*repetitions: int = 1*, *volume: float = None*, *location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None*, *rate: float = 1.0*) → opentrons.protocol_api.contexts.InstrumentContext
Mix a volume of liquid (uL) using this pipette. If no location is specified, the pipette will mix from its current position. If no volume is passed, `mix` will default to the pipette's *max_volume* (page 103).

Parameters

- **repetitions** – how many times the pipette should mix (default: 1)
- **volume** – number of microlitres to mix (default: *max_volume* (page 103))
- **location** (*types.Location* (page 118)) – a Well or a position relative to well. e.g, *plate.rows()[0][0].bottom()*
- **rate** – Set plunger speed for this mix, where, `speed = rate * (aspirate_speed or dispense_speed)`

Raises **NoTipAttachedError** – If no tip is attached to the pipette.

Returns This instance

Note:

All the arguments to `mix` are optional; however, if you do not want to specify one of them, all arguments after that one should be keyword arguments. For instance, if you do not want to specify volume, you would call `pipette.mix(1,location=wellplate['A1'])`. If you do not want to specify repetitions, you would call `pipette.mix(volume=10,location=wellplate['A1'])`. Unlike previous API versions, `mix` will not attempt to guess your

inputs; the first argument will always be interpreted as `repetitions`, the second as `volume`, and the third as `location` unless you use keywords.

---

model
The model string for the pipette (e.g. 'p300_single_v1.3')

mount
Return the name of the mount this pipette is attached to

move_to(*location: opentrons.types.Location*, *force_direct: bool = False*, *minimum_z_height: float = None*, *speed: float = None*) → opentrons.protocol_api.contexts.InstrumentContext
Move the instrument.

Parameters

- **location** (*types.Location* (page 118)) – The location to move to.
- **force_direct** – If set to true, move directly to destination without arc motion.
- **minimum_z_height** – When specified, this Z margin is able to raise (but never lower) the mid-arc height.
- **speed** – The speed at which to move. By default, *InstrumentContext.default_speed* (page 101). This controls the straight linear speed of the motion; to limit individual axis speeds, you can use *ProtocolContext.max_speeds* (page 98).

name
The name string for the pipette (e.g. 'p300_single')

pick_up_current
The current (amperes) the pipette mount's motor will use while picking up a tip. Specified in amps.

pick_up_tip(*location: typing.Union[opentrons.types.Location, opentrons.protocol_api.labware.Well] = None*, *presses: int = None*, *increment: float = 1.0*) → opentrons.protocol_api.contexts.InstrumentContext
Pick up a tip for the pipette to run liquid-handling commands with

If no location is passed, the Pipette will pick up the next available tip in its *InstrumentContext.tip_racks* (page 106) list.

The tip to pick up can be manually specified with the *location* argument. The *location* argument can be specified in several ways:

- If the only thing to specify is which well from which to pick up a tip, *location* can be a *Well* (page 111). For instance, if you have a tip rack in a variable called *tiprack*, you can pick up a specific tip from it with `instr.pick_up_tip(tiprack.wells()[0])`. This style of call can be used to make the robot pick up a tip from a tip rack that was not specified when creating the *InstrumentContext* (page 99).
- If the position to move to in the well needs to be specified, for instance to tell the robot to run its pick up tip routine starting closer to or farther from the top of the tip, *location* can be a *types.Location* (page 118); for instance, you can call `instr.pick_up_tip(tiprack.wells()[0].top())`.

Parameters

- **location** (*types.Location* (page 118) or *Well* (page 111) to pick up a tip from.) – The location from which to pick up a tip.
- **presses** (*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).
- **increment** (*float*) – The additional distance to travel on each successive press (e.g.: if *presses=3* and *increment=1.0*, then the first press will travel down into the tip by 3.5mm, the second by 4.5mm, and the third by 5.5mm).

Returns This instance

reset_tipracks()
Reload all tips in each tip rack and reset starting tip

return_tip() → opentrons.protocol_api.contexts.InstrumentContext
If a tip is currently attached to the pipette, then it will return the tip to it's location in the tiprack.

It will not reset tip tracking so the well flag will remain False.

Returns This instance

speed
The speeds (in mm/s) configured for the pipette plunger.

This is an object with attributes `aspirate`, `dispense`, and `blow_out` holding the plunger speeds for the corresponding operation.

Note: This property is equivalent to *flow_rate* (page 103); the only difference is the units in which this property is specified. Specifying this attribute uses the units of the linear speed of the plunger inside

the pipette, while *flow_rate* (page 103) uses the units of the volumetric flow rate of liquid into or out of the tip. Because *speed* (page 105) and *flow_rate* (page 103) modify the same values, setting one will override the other.

For instance, to set the plunger speed during an aspirate action, do

```
instrument.speed.aspirate = 50
```

starting_tip
The starting tip from which the pipette pick up

tip_racks
The tip racks that have been linked to this pipette.

This is the property used to determine which tips to pick up next when calling *pick_up_tip()* (page 105) without arguments.

touch_tip(*location: opentrons.protocol_api.labware.Well = None*, *radius: float = 1.0*, *v_offset: float = -1.0*, *speed: float = 60.0*) → opentrons.protocol_api.contexts.InstrumentContext
Touch the pipette tip to the sides of a well, with the intent of removing left-over droplets

Parameters

- **location** (*Well* (page 111) or None) – If no location is passed, pipette will touch tip at current well's edges
- **radius** (*float*) – Describes the proportion of the target well's radius. When *radius=1.0*, the pipette tip will move to the edge of the target well; when *radius=0.5*, it will move to 50% of the well's radius. Default: 1.0 (100%)
- **v_offset** (*float*) – The offset in mm from the top of the well to touch tip A positive offset moves the tip higher above the well, while a negative offset moves it lower into the well Default: -1.0 mm
- **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

Raises

- **NoTipAttachedError** – if no tip is attached to the pipette
- **RuntimeError** – If no location is specified and location cache is None. This should happen if *touch_tip* is called without first calling a method that takes a location (eg, *aspirate()* (page 100), *dispense()* (page 102))

Returns This instance

Note: This is behavior change from legacy API (which accepts any *Placeable* (page 55) as the `location` parameter)

---

transfer(*volume: typing.Union[float, typing.Sequence[float]]*, *source*, *dest*, *trash=True*, ***kwargs*) → opentrons.protocol_api.contexts.InstrumentContext
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 *InstrumentContext* (page 99) commands, like *aspirate()* (page 100) and *dispense()* (page 102), designed to make protocol writing easier at the cost of specificity.

Parameters

- **volume** – The amount of volume to aspirate from each source and dispense to each destination. If volume 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** – A single well or a list of wells from where liquid will be aspirated.
- **dest** – A single well or a list of wells where liquid will be dispensed to.
- ****kwargs** – See below

Keyword Arguments

- *new_tip* (`string`) –
  - 'never': no tips will be picked up or dropped during transfer
  - 'once': (default) a single tip will be used for all commands.
  - 'always': use a new tip for each transfer.
- *trash* (`boolean`) – If *True* (default behavior), tips will be dropped in the trash container attached this *Pipette*. If *False* tips will be returned to tiprack.
- *touch_tip* (`boolean`) – If *True*, a *touch_tip()* (page 106) will occur following each *aspirate()* (page 100) and *dispense()* (page 102). If set to *False* (default behavior), no *touch_tip()* (page 106) will occur.
- *blow_out* (`boolean`) – If *True*, a *blow_out()* (page 101) will occur following each *dispense()* (page 102), but only if the pipette has no liquid left in it. If set to *False* (default), no *blow_out()* (page 101) will occur.
- *mix_before* (`tuple`) – The tuple, if specified, gives the amount of volume to *mix()* (page 104) preceding each *aspirate()* (page 100) during the transfer. The tuple is interpreted as (repetitions, volume).
- *mix_after* (`tuple`) – The tuple, if specified, gives the amount of volume to *mix()* (page 104) after each *dispense()* (page 102) during the transfer. The tuple is interpreted as (repetitions, volume).
- *disposal_volume* (`float`) – (*distribute()* (page 102) only) Volume of liquid to be disposed off after distributing. When dispensing multiple times from the same tip, it is recommended to aspirate an extra amount of liquid to be disposed off after distributing.
- *carryover* (`boolean`) – If *True* (default), any *volume* that exceeds the maximum volume of this Pipette will be split into multiple smaller volumes.
- *gradient* (`lambda`) – Function for calculating the curve used for gradient volumes. When *volume* is a tuple of length 2, its 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 This instance

trash_container

The trash container associated with this pipette.

This is the property used to determine where to drop tips and blow out liquids when calling *drop_tip()* (page 102) or *blow_out()* (page 101) without arguments.

type

One of *'single'* or *'multi'*.

well_bottom_clearance

The distance above the bottom of a well to aspirate or dispense.

This is an object with attributes `aspirate` and `dispense`, describing the default heights of the corresponding operation. The default is 1.0mm for both aspirate and dispense.

When *aspirate()* (page 100) or *dispense()* (page 102) is given a *Well* (page 111) rather than a full *Location* (page 118), the robot will move this distance above the bottom of the well to aspirate or dispense.

To change, set the corresponding attribute. For instance,

```
instr.well_bottom_clearance.aspirate = 1
```

Labware and Wells

opentrons.protocol_api.labware: classes and functions for labware handling

This module provides things like *Labware* (page 108), *Well* (page 111), and *ModuleGeometry* (page 111) to encapsulate labware instances used in protocols and their wells. It also provides helper functions to load and save labware and labware calibration offsets. It contains all the code necessary to transform from labware symbolic points (such as "well a1 of an opentrons tiprack") to points in deck coordinates.

class `opentrons.protocol_api.labware.`**Labware**(*definition: dict*, *parent: opentrons.types.Location*, *label: str = None*) → None

This class represents a labware, such as a PCR plate, a tube rack, trough, tip rack, etc. It defines the physical geometry of the labware, and provides methods for accessing wells within the labware.

It is commonly created by calling `ProtocolContext.load_labware()`.

To access a labware's wells, you can use its well accessor methods: *wells_by_name()* (page 110), *wells()* (page 110), *columns()* (page 108), *rows()* (page 110), *rows_by_name()* (page 110), and *columns_by_name()* (page 108). You can also use an instance of a labware as a Python dictionary, accessing wells by their names. The following example shows how to use all of these methods to access well A1:

```
labware = context.load_labware('corning_96_wellplate_360ul_flat', 1)
labware['A1']
labware.wells_by_name()['A1']
labware.wells()[0]
labware.rows()[0][0]
labware.columns()[0][0]
labware.rows_by_name()['A'][0]
labware.columns_by_name()[0][0]
```

columns(**args*) → typing.List[typing.List[opentrons.protocol_api.labware.Well]]

Accessor function used to navigate through a labware by column.

With indexing one can treat it as a typical python nested list. To access row A for example, simply write: labware.columns()[0] This will output ['A1', 'B1', 'C1', 'D1'...].

Note that this method takes args for backward-compatibility, but use of args is deprecated and will be removed in future versions. Args can be either strings or integers, but must all be the same type (e.g.: *self.columns(1, 4, 8)* or *self.columns('1', '2')*, but *self.columns('1', 4)* is invalid.

Returns A list of column lists

columns_by_name() → typing.Dict[str, typing.List[opentrons.protocol_api.labware.Well]]
Accessor function used to navigate through a labware by column name.

With indexing one can treat it as a typical python dictionary. To access row A for example, simply write: labware.columns_by_name()['1'] This will output ['A1', 'B1', 'C1', 'D1'...].

Returns Dictionary of Well lists keyed by column name

highest_z
The z-coordinate of the tallest single point anywhere on the labware.

This is drawn from the 'dimensions'/'zDimension' elements of the labware definition and takes into account the calibration offset.

name
The canonical name of the labware, which is used to load it

next_tip(*num_tips: int = 1*, *starting_tip: opentrons.protocol_api.labware.Well = None*) → typing.Union[opentrons.protocol_api.labware.Well, NoneType]
Find the next valid well for pick-up.

Determines the next valid start tip from which to retrieve the specified number of tips. There must be at least *num_tips* sequential wells for which all wells have tips, in the same column.

Parameters **num_tips** (*int*) – target number of sequential tips in the same column

Returns the *Well* (page 111) meeting the target criteria, or None

parameters
Internal properties of a labware including type and quirks

parent
The parent of this labware. Usually a slot name.

previous_tip(*num_tips: int = 1*) → typing.Union[opentrons.protocol_api.labware.Well, NoneType]
Find the best well to drop a tip in.

This is the well from which the last tip was picked up, if there's room. It can be used to return tips to the tip tracker.

Parameters **num_tips** (*int*) – target number of tips to return, sequential in a column

Returns The *Well* (page 111) meeting the target criteria, or None

quirks
Quirks specific to this labware.

reset()
Reset all tips in a tiprack

return_tips(*start_well: opentrons.protocol_api.labware.Well*, *num_channels: int = 1*)
Re-adds tips to the tip tracker

This method should be called when a tip is dropped in a tiprack. It should be called with *num_channels=1* or *num_channels=8* for single- and multi-channel respectively. If returning more than one channel, this method will automatically determine which tips are returned based on the start well, the number of channels, and the tiprack geometry.

Note that unlike *use_tips()* (page 110), calling this method in a way that would drop tips into wells with tips in them will raise an exception; this should only be called on a valid return of *previous_tip()* (page 109).

Parameters

- **start_well** (*Well* (page 111)) – The *Well* (page 111) into which to return a tip.

- **num_channels** (*int*) – The number of channels for the current pipette

rows(**args*) → typing.List[typing.List[opentrons.protocol_api.labware.Well]]
Accessor function used to navigate through a labware by row.

With indexing one can treat it as a typical python nested list. To access row A for example, simply write: labware.rows()[0]. This will output ['A1', 'A2', 'A3', 'A4'...]

Note that this method takes args for backward-compatibility, but use of args is deprecated and will be removed in future versions. Args can be either strings or integers, but must all be the same type (e.g.: *self.rows(1, 4, 8)* or *self.rows('A', 'B')*, but *self.rows('A', 4)* is invalid.

Returns A list of row lists

rows_by_name() → typing.Dict[str, typing.List[opentrons.protocol_api.labware.Well]]
Accessor function used to navigate through a labware by row name.

With indexing one can treat it as a typical python dictionary. To access row A for example, simply write: labware.rows_by_name()['A'] This will output ['A1', 'A2', 'A3', 'A4'...].

Returns Dictionary of Well lists keyed by row name

set_calibration(*delta: opentrons.types.Point*)
Called by save calibration in order to update the offset on the object.

uri
A string fully identifying the labware.

Returns The uri, `"namespace/loadname/version"`

use_tips(*start_well: opentrons.protocol_api.labware.Well*, *num_channels: int = 1*)
Removes tips from the tip tracker.

This method should be called when a tip is picked up. Generally, it will be called with *num_channels=1* or *num_channels=8* for single- and multi-channel respectively. If picking up with more than one channel, this method will automatically determine which tips are used based on the start well, the number of channels, and the geometry of the tiprack.

Parameters

- **start_well** (*Well* (page 111)) – The *Well* (page 111) from which to pick up a tip. For a single-channel pipette, this is the well to send the pipette to. For a multi-channel pipette, this is the well to send the back-most nozzle of the pipette to.
- **num_channels** (*int*) – The number of channels for the current pipette

well(*idx*) → opentrons.protocol_api.labware.Well
Deprecated—use result of *wells* or *wells_by_name*

wells(**args*) → typing.List[opentrons.protocol_api.labware.Well]
Accessor function used to generate a list of wells in top -> down, left -> right order. This is representative of moving down *rows* and across *columns* (e.g. 'A1', 'B1', 'C1'...'A2', 'B2', 'C2')

With indexing one can treat it as a typical python list. To access well A1, for example, simply write: labware.wells()[0]

Note that this method takes args for backward-compatibility, but use of args is deprecated and will be removed in future versions. Args can be either strings or integers, but must all be the same type (e.g.: *self.wells(1, 4, 8)* or *self.wells('A1', 'B2')*, but *self.wells('A1', 4)* is invalid.

Returns Ordered list of all wells in a labware

wells_by_name() → typing.Dict[str, opentrons.protocol_api.labware.Well]
Accessor function used to create a look-up table of Wells by name.

With indexing one can treat it as a typical python dictionary whose keys are well names. To access well A1, for example, simply write: labware.wells_by_name()['A1']

Returns Dictionary of well objects keyed by well name

class opentrons.protocol_api.labware.**ModuleGeometry**(*definition: dict*, *parent: opentrons.types.Location*) → None

This class represents an active peripheral, such as an Opentrons Magnetic Module, Temperature Module or Thermocycler Module. It defines the physical geometry of the device (primarily the offset that modifies the position of the labware mounted on top of it).

labware_offset

return – a *Point* (page 119) representing the transformation between the critical point of the module and the critical point of its contained labware

location

return – a *Location* (page 118) representing the top of the module

class opentrons.protocol_api.labware.**Well**(*well_props: dict*, *parent: opentrons.types.Location*, *display_name: str*, *has_tip: bool*) → None

The Well class represents a single well in a *Labware* (page 108)

It provides functions to return positions used in operations on the well such as *top()* (page 111), *bottom()* (page 111)

bottom(*z: float = 0.0*) → opentrons.types.Location

Parameters **z** – the z distance in mm

Returns a Point corresponding to the absolute position of the bottom-center of the well (with the front-left corner of slot 1 as (0,0,0)). If z is specified, returns a point offset by z mm from bottom-center

center() → opentrons.types.Location

Returns a Point corresponding to the absolute position of the center

of the well relative to the deck (with the front-left corner of slot 1 as (0,0,0))

top(*z: float = 0.0*) → opentrons.types.Location

Parameters **z** – the z distance in mm

Returns a Point corresponding to the absolute position of the top-center of the well relative to the deck (with the front-left corner of slot 1 as (0,0,0)). If z is specified, returns a point offset by z mm from top-center

class opentrons.protocol_api.labware.**WellShape**

An enumeration.

opentrons.protocol_api.labware.**clear_calibrations**()

Delete all calibration files for labware. This includes deleting tip-length data for tipracks.

opentrons.protocol_api.labware.**get_labware_definition**(*load_name: str*, *namespace: str = None*, *version: int = None*, *bundled_defs: typing.Dict[str, typing.Dict[str, typing.Any]] = None*, *extra_defs: typing.Dict[str, typing.Dict[str, typing.Any]] = None*) → typing.Dict[str, typing.Any]

Look up and return a definition by load_name + namespace + version and return it or raise an exception

Parameters

- **load_name** (*str*) – corresponds to 'loadName' key in definition
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.
- **bundled_defs** – A bundle of labware definitions to exlusively use for finding labware definitions, if specified
- **extra_defs** – An extra set of definitions (in addition to the system definitions) in which to search

opentrons.protocol_api.labware.**load**(*load_name: str*, *parent: opentrons.types.Location*, *label: str = None*, *namespace: str = None*, *version: int = 1*, *bundled_defs: typing.Dict[str, typing.Dict[str, typing.Any]] = None*, *extra_defs: typing.Dict[str, typing.Dict[str, typing.Any]] = None*) → opentrons.protocol_api.labware.Labware

Return a labware object constructed from a labware definition dict looked up by name (definition must have been previously stored locally on the robot)

Parameters

- **load_name** – A string to use for looking up a labware definition previously saved to disc. The definition file must have been saved in a known location
- **parent** – A *Location* (page 118) representing the location where the front and left most point of the outside of labware is (often the front-left corner of a slot on the deck).
- **label** (*str*) – An optional label that will override the labware's display name from its definition
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.
- **bundled_defs** – If specified, a mapping of labware names to labware definitions. Only the bundle will be searched for definitions.
- **extra_defs** – If specified, a mapping of labware names to labware definitions. If no bundle is passed, these definitions will also be searched.

opentrons.protocol_api.labware.**load_calibration**(*labware: opentrons.protocol_api.labware.Labware*)

Look up a calibration if it exists and apply it to the given labware.

opentrons.protocol_api.labware.**load_from_definition**(*definition: dict*, *parent: opentrons.types.Location*, *label: str = None*) → opentrons.protocol_api.labware.Labware

Return a labware object constructed from a provided labware definition dict

Parameters

- **definition** – A dict representing all required data for a labware, including metadata such as the display name of the labware, a definition of the order to iterate over wells, the shape of wells (shape, physical dimensions, etc), and so on. The correct shape of this definition is governed by the "labware-designer" project in the Opentrons/opentrons repo.

- **parent** – A *Location* (page 118) representing the location where the front and left most point of the outside of labware is (often the front-left corner of a slot on the deck).
- **label** (*str*) – An optional label that will override the labware's display name from its definition

opentrons.protocol_api.labware.**load_module**(*name: str, parent: opentrons.types.Location*) → opentrons.protocol_api.labware.ModuleGeometry

Return a *ModuleGeometry* (page 111) object from a definition looked up by name.

Parameters

- **name** – A string to use for looking up the definition. The string must be present as a top-level key in module/definitions/{moduleDefinitionVersion}.json.
- **parent** – A *Location* (page 118) representing the location where the front and left most point of the outside of the module is (often the front-left corner of a slot on the deck).

opentrons.protocol_api.labware.**load_module_from_definition**(*definition: dict, parent: opentrons.types.Location*) → opentrons.protocol_api.labware.ModuleGeometry

Return a *ModuleGeometry* (page 111) object from a specified definition

Parameters

- **definition** – A dict representing all required data for a module's geometry.
- **parent** – A *Location* (page 118) representing the location where the front and left most point of the outside of the module is (often the front-left corner of a slot on the deck).

opentrons.protocol_api.labware.**quirks_from_any_parent**(*loc: typing.Union[opentrons.protocol_api.labware.Labware, opentrons.protocol_api.labware.Well, str, opentrons.protocol_api.labware.ModuleGeometry, NoneType]*) → typing.List[str]

Walk the tree of wells and labwares and extract quirks

opentrons.protocol_api.labware.**save_calibration**(*labware: opentrons.protocol_api.labware.Labware, delta: opentrons.types.Point*)

Function to be used whenever an updated delta is found for the first well of a given labware. If an offset file does not exist, create the file using labware id as the filename. If the file does exist, load it and modify the delta and the lastModified fields under the "default" key.

opentrons.protocol_api.labware.**save_definition**(*labware_def: typing.Dict[str, typing.Any], force: bool = False*) → None

Save a labware definition

Parameters

- **labware_def** – A deserialized JSON labware definition
- **force** (*bool*) – If true, overwrite an existing definition if found. Cannot overwrite Opentrons definitions.

opentrons.protocol_api.labware.**save_tip_length**(*labware: opentrons.protocol_api.labware.Labware*, *length: float*)

Function to be used whenever an updated tip length is found for of a given tip rack. If an offset file does not exist, create the file using labware id as the filename. If the file does exist, load it and modify the length and the lastModified fields under the "tipLength" key.

opentrons.protocol_api.labware.**uri_from_definition**(*definition: typing.Dict[str, typing.Any]*, *delimiter='/'*) → str

Build a labware URI from its definition.

A labware URI is a string that uniquely specifies a labware definition.

Returns str The URI.

opentrons.protocol_api.labware.**uri_from_details**(*namespace: str*, *load_name: str*, *version: str*, *delimiter='/'*) → str

Build a labware URI from its details.

A labware URI is a string that uniquely specifies a labware definition.

Returns str The URI.

opentrons.protocol_api.labware.**verify_definition**(*contents: AnyStr*) → typing.Dict[str, typing.Any]

Verify that an input string is a labware definition and return it.

If the definition is invalid, an exception is raised; otherwise parse the json and return the valid definition.

Raises

- **json.JsonDecodeError** – If the definition is not valid json
- **jsonschema.ValidationError** – If the definition is not valid.

Returns The parsed definition

Modules

class opentrons.protocol_api.contexts.**TemperatureModuleContext**(*ctx: opentrons.protocol_api.contexts.ProtocolContext*, *hw_module: opentrons.hardware_control.modules.tempdeck.T*, *geometry: opentrons.protocol_api.labware.ModuleGeometry*, *loop: asyncio.events.AbstractEventLoop*) → None

An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 97) using: `ctx.load_module('Temperature Module',slot_number)`.

A minimal protocol with a Temperature module would look like this:

Note: In order to prevent physical obstruction of other slots, place the Temperature Module in a slot on the horizontal edges of the deck (such as 1, 4, 7, or 10 on the left or 3, 6, or 7 on the right), with the USB cable and power cord pointing away from the deck.

---

deactivate()
Stop heating (or cooling) and turn off the fan.

geometry
The object representing the module as an item on the deck

Returns ModuleGeometry

labware
The labware (if any) present on this module.

load_labware(*name: str*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters **name** – The name of the labware object.

Returns The initialized and loaded labware object.

load_labware_object(*labware: opentrons.protocol_api.labware.Labware*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters **labware** – The labware object. This object should be already initialized and its parent should be set to this module's geometry. To initialize and load a labware onto the module in one step, see `load_labware()`.

Returns The properly-linked labware object

set_temperature(*celsius: float*)
Set the target temperature, in C.

Must be between 4 and 95C based on Opentrons QA.

Parameters **celsius** – The target temperature, in C

target
Current target temperature in C

temperature
Current temperature in C

wait_for_temp()
Block until the module reaches its setpoint.

class opentrons.protocol_api.contexts.**MagneticModuleContext**(*ctx: opentrons.protocol_api.contexts.ProtocolContext*, *hw_module: opentrons.hardware_control.modules.magdeck.MagD*, *geometry: opentrons.protocol_api.labware.ModuleGeometry*, *loop: asyncio.events.AbstractEventLoop*) → None
An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 97).

calibrate()
Calibrate the Magnetic Module.

The calibration is used to establish the position of the lawbare on top of the magnetic module.

disengage()
Lower the magnets back into the Magnetic Module.

engage(*height: float = None*, *offset: float = None*)
Raise the Magnetic Module's magnets.

The destination of the magnets can be specified in several different ways, based on internally stored default heights for labware:

- If neither *height* nor *offset* is specified, the magnets will raise to a reasonable default height based on the specified labware.
- If *height* is specified, it should be a distance in mm from the home position of the magnets.
- If *offset* is specified, it should be an offset in mm from the default position. A positive number moves the magnets higher and a negative number moves the magnets lower.

Only certain labwares have defined engage heights for the Magnetic Module. If a labware that does not have a defined engage height is loaded on the Magnetic Module (or if no labware is loaded), then *height* must be specified.

Parameters

- **height** – The height to raise the magnets to, in mm from home.
- **offset** – An offset relative to the default height for the labware in mm

geometry
The object representing the module as an item on the deck

Returns ModuleGeometry

labware
The labware (if any) present on this module.

load_labware(*name: str*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters **name** – The name of the labware object.

Returns The initialized and loaded labware object.

load_labware_object(*labware: opentrons.protocol_api.labware.Labware*) → opentrons.protocol_api.labware.Labware
Load labware onto a Magnetic Module, checking if it is compatible

status
The status of the module. either 'engaged' or 'disengaged'

class opentrons.protocol_api.contexts.**ThermocyclerContext**(*ctx: opentrons.protocol_api.contexts.ProtocolContext*, *hw_module: opentrons.hardware_control.modules.thermocycler.Ther*
geometry: opentrons.protocol_api.labware.ThermocyclerGeometry
loop: asyncio.events.AbstractEventLoop)
→ None

An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 97).

block_target_temperature
Target temperature in degrees C

block_temperature
Current temperature in degrees C

close_lid()
Closes the lid

deactivate()
Turn off the well block, and heated lid

deactivate_block()
Turn off the well block

deactivate_lid()
Turn off the heated lid

execute_profile(*steps: typing.List[typing.Dict[str, float]], repetitions: int*)
Execute a Thermocycler Profile defined as a cycle of `steps` to repeat for a given number of `repetitions`

Parameters

- **steps** – List of unique steps that make up a single cycle. Each list item should be a dictionary that maps to the parameters of the `set_block_temperature()` method with keys 'temperature', 'hold_time_seconds', and 'hold_time_minutes'.
- **repetitions** – The number of times to repeat the cycled steps.

geometry
The object representing the module as an item on the deck

Returns ModuleGeometry

labware
The labware (if any) present on this module.

lid_position
Lid open/close status string

lid_target_temperature
Target temperature in degrees C

lid_temperature
Current temperature in degrees C

load_labware(*name: str*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters **name** – The name of the labware object.

Returns The initialized and loaded labware object.

load_labware_object(*labware: opentrons.protocol_api.labware.Labware*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters **labware** – The labware object. This object should be already initialized and its parent should be set to this module's geometry. To initialize and load a labware onto the module in one step, see `load_labware()`.

Returns The properly-linked labware object

open_lid()
Opens the lid

set_block_temperature(*temperature: float*, *hold_time_seconds: float = None*, *hold_time_minutes: float = None*, *ramp_rate: float = None*)
Set the target temperature for the well block, in °C.

Valid operational range yet to be determined. :param temperature: The target temperature, in °C. :param hold_time_minutes: The number of minutes to hold, after reaching

`temperature`, before proceeding to the next command.

Parameters

- **hold_time_seconds** – The number of seconds to hold, after reaching `temperature`, before proceeding to the next command. If `hold_time_minutes` and `hold_time_seconds` are not specified, the Thermocycler will proceed to the next command after `temperature` is reached.
- **ramp_rate** – The target rate of temperature change, in °C/sec. If `ramp_rate` is not specified, it will default to the maximum ramp rate as defined in the device configuration.

set_lid_temperature(*temperature: float*)
Set the target temperature for the heated lid, in °C.

Parameters **temperature** – The target temperature, in °C clamped to the range 20°C to 105°C.

Useful Types and Definitions

class `opentrons.types.`**Location**
A location to target as a motion in the protocol-api.

The location contains a *Point* (page 119) (in protocol-api-deck-coordinates) and possibly an associated *Labware* (page 108) or *Well* (page 111) instance.

It should rarely be constructed directly by the user; rather, it is the return type of most *Well* (page 111) accessors like *Well.top()* (page 111) and is passed directly into a method like `InstrumentContext.aspirate()`.

Warning: The *labware* (page 118) attribute of this class is used by the protocol API internals to, among other things, determine safe heights to retract the instruments to when moving between locations. If constructing an instance of this class manually, be sure to either specify *None* as the labware (so the robot does its worst case retraction) or specify the correct labware for the *point* (page 119) attribute.

Warning: The `==` operation compares both the position and associated labware. If you only need to compare locations, compare the *point* (page 119) of each item.

labware
Alias for field number 1

move(*point: opentrons.types.Point*) → opentrons.types.Location
Alter the point stored in the location while preserving the labware.

This returns a new Location and does not alter the current one. It should be used like

```
>>> loc = Location(Point(1, 1, 1), 'Hi')
>>> new_loc = loc.move(Point(1, 1, 1))
>>> assert loc_2.point == Point(2, 2, 2)  # True
>>> assert loc.point == Point(1, 1, 1)  # True
```

point
Alias for field number 0

class opentrons.types.**Mount**
An enumeration.

exception opentrons.types.**PipetteNotAttachedError**
An error raised if a pipette is accessed that is not attached

class opentrons.types.**Point**(*x*, *y*, *z*)

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

class opentrons.types.**TransferTipPolicy**
An enumeration.

Deck Coordinates

The OT2's base coordinate system is known as deck coordinates. This coordinate system is referenced frequently through the API. It is a right-handed coordinate system always specified in mm, with *(0, 0, 0)* at the front left of the robot. $+x$ is to the right, $+y$ is to the back, and $+z$ is up.

Note that there are technically two z axes, one for each pipette mount. In these terms, z is the axis of the left pipette mount and a is the axis of the right pipette mount. These are obscured by the API's habit of defining motion commands on a per-pipette basis; the pipettes internally select the correct z axis to move. This is also true of the pipette plunger axes, b (left) and c (right).

When locations are specified to functions like *opentrons.protocol_api.contexts.InstrumentContext.move_to()* (page 104), in addition to being an instance of *opentrons.protocol_api.labware.Well* (page 111) they may define coordinates in this deck coordinate space. These coordinates can be specified either as a standard python `tuple` of three floats, or as an instance of the `collections.namedtuple` *opentrons.types.Point* (page 119), which can be created in the same way.

Examples

All examples on this page use a `'corning_96_wellplate_360ul_flat'` (an ANSI standard 96-well plate[40]) in slot 1, and two `'opentrons_96_tiprack_300ul'` (the Opentrons standard 300 µL tiprack[41]) in slots 2 and 3. They also require a P300 Single attached to the right mount. Some examples also use a `'usascientific_12_reservoir_22ml'` (a USA Scientific 12-row trough[42]) in slot 4.

---

[40] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[41] https://labware.opentrons.com/opentrons_96_tiprack_300ul
[42] https://labware.opentrons.com/usascientific_12_reservoir_22ml

Basic Transfer

Moving 100uL from one well to another:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])

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

This accomplishes the same thing as the following basic commands:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])

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

Loops

Loops in Python allow 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:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    trough = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])
    # distribute 20uL from trough:A1 -> plate:row:1
    # distribute 20uL from trough:A2 -> plate:row:2
    # etc...

    # range() starts at 0 and stops before 8, creating a range of 0-7
    for i in range(8):
        p300.distribute(200, trough.wells()[i], plate.rows()[i])
```

Multiple Air Gaps

The OT-2 pipettes 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 an air gap between each sample.

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    trough = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])
    p300.pick_up_tip()

    for well in trough.wells():
        p300.aspirate(35, well)
        p300.air_gap(10)
        p300.dispense(plate['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.

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    tiprack_2 = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    trough = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1,␣
↪tiprack_2])
    p300.distribute(50, trough['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[0])

        # dilute the sample down the column
        p300.transfer(
            30, row.wells()[1:11], row.wells()[2:],
            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.

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    tiprack_2 = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    trough = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1,␣
↪tiprack_2])

    # 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['A12'], plate.wells())
```

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

Running Protocols Directly On The Robot

Sometimes, you may write a protocol that is not suitable for execution through the Opentrons App. Perhaps it requires user input; perhaps it needs to do a lot of things it cannot do when being simulated. There are two ways to run a protocol on the robot without using the Opentrons app.

Jupyter Notebook

The robot runs a Jupyter notebook server that you can connect to with your web browser. This is a convenient environment in which to write and debug protocols, since you can define different parts of your protocol in different notebook cells and run only part of the protocol at a given time.

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.

Protocol Structure

To take advantage of Jupyter's ability to run only parts of your protocol, you have to restructure it slightly - turn it inside out. Rather than writing a single `run` function that contains all your protocol logic, you can use the function `opentrons.execute.get_protocol_api()`:

```
>>> import opentrons.execute
>>> protocol = opentrons.execute.get_protocol_api()
```

This returns the same kind of object - a *`ProtocolContext`* (page 95) - that is passed into your protocol's `run` function when you upload your protocol in the Opentrons App. Full documentation on the capabilities and use of the *`ProtocolContext`* (page 95) object is available in the other sections of this guide - *Overview* (page 125), *Pipettes* (page 60), *Building Block Commands* (page 76), *Complex Commands* (page 82), *Labware* (page 64), and *Hardware Modules* (page 68); a full list of all available attributes and methods is available in *API Version 2 Reference* (page 95).

Whenever you call *get_protocol_api*, the robot will update its cache of attached instruments and modules. You can call *get_protocol_api* repeatedly; it will return an entirely new *`ProtocolContext`* (page 95) each time, without any labware loaded or any instruments established. This can be a good way to reset the state of the system, if you accidentally loaded in the wrong labware.

Now that you have a *`ProtocolContext`* (page 95), you call all its methods just as you would in a protocol, without the encompassing `run` function, just like if you were prototyping a plotting or pandas script for later use.

Running A Previously-Written Protocol

If you have a protocol that you have already written, that is defined in a `run` function, and that you don't want to modify, you can run it directly in Jupyter. Copy the protocol into a cell and execute it - this won't cause the robot to move, it just makes the function available. Then, call the `run` function you just defined, and give it a *`ProtocolContext`* (page 95):

```
>>> import opentrons.execute
>>> from opentrons import protocol_api
>>> def run(protocol: protocol_api.ProtocolContext):
...     # the contents of your protocol are here...
...
>>> protocol = opentrons.execute.get_protocol_api()
>>> run(protocol)  # your protocol will now run
```

Command Line

The robot's command line is accessible either by creating a new terminal in Jupyter or by using SSH to access its terminal. Sometimes, you may want to run a protocol on the robot terminal directly, without using the Opentrons App or the robot's Jupyter notebook. To do this, use the command line program `opentrons_execute`:

```
# opentrons_execute /data/my_protocol.py
```

You can access help on the usage of `opentrons_execute` by calling `opentrons_execute --help`. This script has a couple options to let you customize what it prints out when you run it. By default, it will print out the same runlog you see in the Opentrons App when running a protocol, as it executes; it will also print out internal logs at level `warning` or above. Both of these behaviors can be changed.

Bundling Protocols

Warning: Bundled protocols are a beta feature. The only way to create them is with the `opentrons_simulate` script. The format of the bundle files themselves is subject to change. This is a feature you should use with care. Only very limited support from Opentrons is available for this beta feature.

Bundled protocols are zip files containing

1. an APIv2 protocol
2. Definitions for all required labware for the protocol, including the fixed trash
3. Additional data files that will be made available to the protocol

Bundled protocols may be uploaded through the Opentrons App in their zipped form, just like normal protocols. They may be simulated with `opentrons_simulate` and executed from the robot command line with `opentrons_execute` just like normal protocols.

The advantage to using bundled protocols is that you can pack in custom labware definitions and custom data files such as CSVs specifying aspiration amounts and locations.

Writing A Bundled Protocol

When you write a bundled protocol, you write a normal APIv2 Python protocol. It may or may not include custom labware or data files. It is written in Python using the same API as any other APIv2 Python protocol.

Bundled protocols have all their labware definitions available to them inside the bundle, including both standard and custom definitions. They are limited to loading labware defined in the bundle; for this reason, **if you change what labware you use in a bundled protocol you must rebundle it**.

Bundled protocols also have any data files they may need available to them inside the bundle. Similarly to labware, if you change what data files you read inside the protocol you should rebundle it.

Bundled protocols are created using `opentrons_simulate`. The protocol must be an APIv2 protocol, and `opentrons_simulate` must be running in APIv2 mode. The easiest way to do this is to specify it with the environment variable `OT_API_FF_useProtocolApi2=1`. You can specify this every time you run `opentrons_simulate` on Linux or Mac, or put it in your shell rc file; on Windows, you can set it in the environment variables dialog.

To bundle, use the `-b` option to `opentrons_simulate`. **If the ‘‘-b‘‘ option is not available, it is because you have not set the APIv2 feature flag**. This will simulate the protocol, then (if successful) bundle the protocol file, all required labware definitions, and any specified data file into a zip suitable for use with the Opentrons app or the `opentrons_execute` script. If you are using custom data files or custom labware definitions, you must ensure that these files and definitions are available to `opentrons_simulate`.

Accessing Custom Labware Definitions

To access a labware definition inside a bundle, use *`ProtocolContext.load_labware()`* (page 97) just like in a normal protocol. To make custom labware definitions available to `opentrons_simulate`, use the `-L` option. By default, any labware definition in the current directory when you run `opentrons_simulate` is available to the protocol.

Accessing Custom Data

Custom data files are made available in *ProtocolContext.bundled_data* (page 95). This is a dictionary mapping the names of data files (without any paths) to their contents, as bytes. If you need the contents of the files as strings, you must decode them with `.decode('utf-8')` (the files are presented in bytes in case they are not text, for instance if they are images or zip files). These can then be read in whatever format you need.

For instance, if a CSV file called `aspirations.csv` is bundled, you can do:

To make a custom data file available to `opentrons_simulate`, use the `-d` option to specify a file.

Executing A Bundled Protocol

Once you have a bundled protocol file (by default, its file extension will be `.ot2.zip`) you can use it without any further specification of labware or data files - they are all bundled inside the file. For instance,

1. You can execute a bundled protocol through the Opentrons App by selecting it in the protocol pane
2. You can execute a bundled protocol on the robot command line by doing `opentrons_execute ./protocol.ot2.zip`
3. You can simulate a bundled protocol on your computer by doing `opentrons_simulate ./protocol.ot2.zip`.

Overview

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 instructions to transfer from well `'A1'` to well `'B1'` that even a computer could understand:

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

Begin the protocol

Add a 96 well plate, and place it in slot '2' of the robot deck
Add a 300 µL tip rack, and place it in slot '1' of the robot deck

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

Transfer 100 µL from the plate's 'A1' well to its 'B2' well
```

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

```
from opentrons import protocol_api

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

# protocol run function. the part after the colon lets your editor know
```

```
# where to look for autocomplete suggestions
def run(protocol: protocol_api.ProtocolContext):

    # labware
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', '2')
    tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', '1')

    # pipettes
    left_pipette = protocol.load_instrument(
         'p300_single', 'left', tip_racks=[tiprack])

    # commands
    left_pipette.pick_up_tip()
    left_pipette.aspirate(100, plate['A1'])
    left_pipette.dispense(100, plate['B2'])
    left_pipette.drop_tip()
```

How it's Organized

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

1. Metadata
2. Run function
3. Labware
4. Pipettes
5. Commands

Metadata

Metadata is a dictionary of data that is read by the server and returned to client applications (such as the Opentrons Run 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 fields, or additional fields as desired (though non-standard fields are not displayed by the Opentrons app).

The Run Function and the Protocol Context

Opentrons API version 2 protocols are structured around a function called `run(protocol)`, defined in code like this:

```
from opentrons import protocol_api

def run(protocol: protocol_api.ProtocolContext):
    pass
```

This function must be named exactly `run` and must take exactly one mandatory argument (its name doesn't matter, but we recommend `protocol` since this argument represents the protocol that the robot will execute).

The function `run` is the container for the code that defines your protocol.

The object `protocol` is the *protocol context*, which represents the robot and its capabilities. It is always an instance of the *`opentrons.protocol_api.contexts.ProtocolContext`* (page 95) class (though you'll never have to instantiate one yourself - it is always passed in to `run()`), and it is tagged as such in the example protocol to allow most editors to give you autocomplete.

The protocol context has two responsibilities:

1. Remember, track, and check the robot's state
2. Expose the functions that make the robot execute actions

The protocol context plays the same role as the `robot`, `labware`, `instruments`, and `modules` objects in past versions of the API, with one important difference: it is only one object; and because it is passed in to your protocol rather than imported, it is possible for the API to be much more rigorous about separating simulation from reality.

The key point is that there is no longer any need to `import opentrons` at the top of every protocol, since the *robot* now *runs the protocol*, rather than the *protocol running the robot*. The example protocol imports the definition of the protocol context to provide editors with autocomplete sources.

Labware

The next step is defining the labware required for your protocol. You must tell the protocol context about what should be present on the deck, and where. You tell the protocol context about labware by calling the method `protocol.load_labware(name,slot)` and saving the result.

The name of a labware is a string that is different for each kind of labware. You can look up labware to add to your protocol on the Opentrons Labware Library[43].

The slot is the labelled location on the deck in which you've placed the labware. The available slots are numbered from 1-11.

Our example protocol above loads a Corning 96 Well Plate[44] in slot 2 (`plate = protocol.load_labware('corning_96_wellplate_360ul_flat',2)`) and an Opentrons 300ul Tiprack[45] in slot 1 (`tiprack = protocol.load_labware('opentrons_96_tiprack_300ul',1)`). These can be referenced later in the protocol as `plate` and `tiprack` respectively. Check out the python docs[46] for further clarification on using variables effectively in your code.

You can find more information about handling labware in the *Labware* (page 64) section.

Pipettes

After defining labware, you define the instruments required for your protocol. You tell the protocol context about which pipettes should be attached, and which slot they should be attached to, by calling the method `protocol.load_instrument(model,mount,tip_racks)` and saving the result.

The `model` of the pipette is the kind of pipette that should be attached; the `mount` is either `"left"` or `"right"`; and `tip_racks` is a list of the objects representing tip racks that this instrument should use. Specifying `tip_racks` is optional, but if you don't then you'll have to manually specify where the instrument should pick up tips from every time you try and pick up a tip.

[43] https://labware.opentrons.com
[44] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[45] https://labware.opentrons.com/opentrons_96_tiprack_300ul
[46] https://docs.python.org/3/index.html

See *Pipettes* (page 60) for more information on creating and working with pipettes.

Our example protocol above loads a P300 Single-channel pipette (`'p300_single'`) in the left mount (`'left'`), and uses the Opentrons 300ul tiprack we loaded previously as a source of tips (`tip_racks=[tiprack]`).

Commands

Once the instruments and labware required for the protocol are defined, the next step is to define the commands that make up the protocol. The most common commands are `aspirate()`, `dispense()`, `pick_up_tip()`, and `drop_tip()`. These and many others are described in the *Building Block Commands* (page 76) and *Complex Commands* (page 82) sections, which go into more detail about the commands and how they work. These commands typically specify which wells of which labware to interact with, using the labware you defined earlier, and are methods of the instruments you created in the pipette section. For instance, in our example protocol, you use the pipette you defined to:

1. Pick up a tip (implicitly from the tiprack you specified in slot 1 and assigned to the pipette): `pipette.pick_up_tip()`
2. Aspirate 100ul from well A1 of the 96 well plate you specified in slot 2: `pipette.aspirate(100,plate['A1'])`
3. Dispense 100ul into well A2 of the 96 well plate you specified in slot 2: `pipette.dispense(100,plate['A2'])`
4. Drop the tip (implicitly into the trash at the back right of the robot's deck): `pipette.drop_tip()`

Python Module Index

O

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z