Opentrons Python API V2 Documentation

Release 3.21.0

Opentrons Labworks

Sep 30, 2020

Contents

The OT-2 Python Protocol 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.

Version 2 of the API is a new way to write Python protocols. It is more reliable, simpler, and better able to be supported. Unlike version 1, it has support for new modules like the Thermocycler. While version 1 will still recieve bug fixes, new features and improvements will land in version 2. For a guide on transitioning your protocols from version 1 to version 2 of the API, see this article on migration[1]. For a more in-depth discussion of why version 2 of the API was developed and what is different about it compared to version 1, see this article on why we wrote API V2[2].

---

1 Getting Started

New to Python? Check out our *Using Python For Protocols* (page 5) page first before continuing. To get a sense of the typical structure of our scripts, take a look at our *Examples* (page 93) page.

To simulate protocols on your laptop, check out *Simulating Your Scripts* (page 7). When you're ready to run your script on a robot, download our latest desktop app[3].

2 Troubleshooting

If you encounter problems using our products please take a look at our support docs[4] or contact our team via intercom on our website at opentrons.com[5].

---

3 Overview

3.1 How it Looks

The design goal of this 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 OT-2 Python Protocol API. It uses version 2.0 of this 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 Python Protocol API, it would look like the following:

[1] http://support.opentrons.com/en/articles/3425727-switching-your-protocols-from-api-version-1-to-version-2
[2] http://support.opentrons.com/en/articles/3418212-opentrons-protocol-api-version-2
[3] https://www.opentrons.com/ot-app
[4] https://support.opentrons.com/en/
[5] https://opentrons.com

```
from opentrons import protocol_api

# metadata
metadata = {
    'protocolName': 'My Protocol',
    'author': 'Name <email@address.com>',
    'description': 'Simple protocol to get started using OT2',
    'apiLevel': '2.7'
}

# 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()
```

3.2 How it's Organized

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

1) Metadata and Version Selection
2) Run function
3) Labware
4) Pipettes
5) Commands

Metadata and Version Selection

Metadata is a dictionary of data that is read by the server and returned to client applications (such as the Opentrons App). Most metadata is not needed to run a protocol, but if present can help the Opentrons App display additional data about the protocol currently being executed. These optional (but recommended) fields are (`"protocolName"`, `"author"`, and `"description"`).

The required element of the metadata is `"apiLevel"`. This must contain a string specifying the major and minor version of the Python Protocol API that your protocol is designed for. For instance, a protocol written for version 2.0 of the Python Protocol API (only launch version of the Protocol API should have in its metadata `"apiLevel": "2.0"`.

For more information on Python Protocol API versioning, see *Versioning* (page 9).

The Run Function and the Protocol Context

Protocols are structured around a function called `run(protocol)`, defined in code like this:

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 58) 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[6].

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[7] in slot 2:

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

- an Opentrons 300µL Tiprack[8] in slot 1:

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

These labware can be referenced later in the protocol as `plate` and `tiprack` respectively. Check out the Python docs[9] for further clarification on using variables effectively in your code.

You can find more information about handling labware in the *Labware* (page 12) section.

[6] https://labware.opentrons.com
[7] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[8] https://labware.opentrons.com/opentrons_96_tiprack_300ul
[9] https://docs.python.org/3/index.html

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.

See *Pipettes* (page 27) 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 300 µL 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 35) and *Complex Commands* (page 45) 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 100 µL from well A1 of the 96 well plate you specified in slot 2: `pipette.aspirate(100, plate['A1'])`
3) Dispense 100 µL 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()`

---

4 Feature Requests

Have an interesting idea or improvement for our software? Create a ticket on GitHub by following these guidelines.[10]

5 Developer's guide

Do you want to contribute to our open-source API? You can find more information on how to be involved here.[11]

5.1 Using Python For Protocols

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 OT-2 Python Protocol API looks like:

[10] https://github.com/Opentrons/opentrons/blob/edge/CONTRIBUTING.md#opening-issues
[11] https://github.com/Opentrons/opentrons/blob/edge/CONTRIBUTING.md

1) Write a Python protocol
2) Test the code for errors
3) Repeat steps 1 & 2
4) Calibrate labware on your OT-2
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 Protocol API (from learnpython.org[12]):

1) Hello World[13]
2) Variables and Types[14]
3) Lists[15]
4) Basic Operators[16]
5) Conditions[17]
6) Loops[18]
7) Functions[19]
8) Dictionaries[20]

After going through the above tutorials, you should have enough of an understanding of Python to work with the Protocol API and start designing your experiments! More detailed information on Python can always be found at the Python docs[21]

Working with Python

Using a popular and free code editor, like Visual Studio Code[22], is a common method for writing Python protocols. Download onto your computer, and you can now write Python scripts.

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

For example, `my_protocol.py`

---

[12] http://www.learnpython.org/
[13] http://www.learnpython.org/en/Hello%2C_World%21
[14] http://www.learnpython.org/en/Variables_and_Types
[15] http://www.learnpython.org/en/Lists
[16] http://www.learnpython.org/en/Basic_Operators
[17] http://www.learnpython.org/en/Conditions
[18] http://www.learnpython.org/en/Loops
[19] http://www.learnpython.org/en/Functions
[20] http://www.learnpython.org/en/Dictionaries
[21] https://docs.python.org/3/index.html
[22] https://code.visualstudio.com/

Simulating Python Protocols

In general, the best way to simulate a protocol is to simply upload it to your OT-2 through the Opentrons App. When you upload a protocol via the app, the OT-2 simulates the protocol and the app displays any errors. However, if you want to simulate protocols without being connected to an OT-2, 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 9), ONLY for your locally-installed notebook) or not.

Non-Jupyter Installation

First, install Python 3.7.6 (Windows x64[23], Windows x86[24], OS X[25]) or higher 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 higher than 3.7.6, you have another version of Python installed; this happens frequently on OS X and sometimes on Windows. We recommend using a tool like pyenv[26] to manage multiple Python versions. This is particularly useful on OS X, which has a built-in install of Python 2.7 that should not be removed.

Once Python is installed, install the opentrons package[27] using `pip`:

```
$ pip install opentrons
```

You should see some output that ends with `Successfully installed opentrons-3.21.0`.

Jupyter Installation

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

```
>>> import sys
>>> !sys.executable -m pip install opentrons
```

Simulating Your Scripts

From the Command Line

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.

[23] https://www.python.org/ftp/python/3.7.6/python-3.7.6-amd64.exe
[24] https://www.python.org/ftp/python/3.7.6/python-3.7.6.exe
[25] https://www.python.org/ftp/python/3.7.6/python-3.7.6-macosx10.6.pkg
[26] https://github.com/pyenv/pyenv
[27] https://pypi.org/project/opentrons/

The simulation script can also be invoked through python:

```
$ python -m opentrons.simulate /path/to/protocol
```

`opentrons_simulate` has several command line options that might be useful. Most options are explained below, but to see all options you can run

```
$ opentrons_simulate --help
```

Using Custom Labware

By default, `opentrons_simulate` will load custom labware definitions from the directory in which you run it. You can change the directory `opentrons_simulate` searches for custom labware with the `--custom-labware-path` option:

```
python.exe -m opentrons.simulate --custom-labware-path="C:\Custom Labware"
```

In the Python Shell

The Opentrons Python package 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()* (page 92):

```
from opentrons.simulate import simulate, format_runlog
# read the file
protocol_file = open('/path/to/protocol.py')
# simulate() the protocol, keeping the runlog
runlog, _bundle = simulate(protocol_file)
# print the runlog
print(format_runlog(runlog))
```

The *opentrons.simulate.simulate()* (page 92) 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 91) turns that list of dictionaries into a human readable string, which is then printed out. For more information on the protocol simulator, see *Simulating Your Scripts* (page 7).

Using Jupyter

In your Jupyter notebook, you can use the Python Protocol API simulator by doing

```
from opentrons import simulate
protocol = simulate.get_protocol_api('2.7')
p300 = protocol.load_instrument('p300_single', 'right')
# ...
```

The `protocol` object, which is an instance of *ProtocolContext* (page 58), is the same thing that gets passed to your protocol's `run` function, but set to simulate rather than control an OT-2. You can call all your protocol's functions on that object.

If you have a full protocol, wrapped inside a `run` function, defined in a Jupyter cell you can also use *opentrons.simulate.simulate()* (page 92) as described above to simulate the protocol.

These instructions also work on the OT-2's Jupyter notebook. This can also be used in the Python interactive shell.

Configuration and Local Storage

The Opentrons Python package uses a folder in your user directory as a place to store and read configuration and changes to its internal data. This location is `~/.opentrons` on Linux or OSX and `C:\Users\%USERNAME%\.opentrons` on Windows.

Robot's Jupyter Notebook

Your OT-2 also has a Jupyter notebook, which you can use to develop and execute protocols. For more information on how to execute protocols using the OT-2's Jupyter notebook, please see *Advanced Control* (page 96). To simulate protocols on the OT-2's Jupyter notebook, use the instructions above.

5.2 Versioning

The OT-2 Python Protocol API has its own versioning system, which is separated from the version of the OT-2 software or of the Opentrons App. This separation allows you to specify the Protocol Api version your protocol requires without being concerned with what OT-2 software versions it will work with, and allows Opentrons to version the Python Protocol API based only on changes that affect protocols.

The API is versioned with a major and minor version, expressed like this: `major.minor`. For instance, major version 2 and minor version 0 is written as `2.0`. Versions are not decimal numbers. Major version 2 and minor version 10 is written as `2.10`, while `2.1` means major version 2 and minor version 1.

Major and Minor Version

The major version of the API is increased whenever there are signficant structural or behavioral changes to protocols. For instance, major version 2 of the API was introduced because protocols must now have a `run` function that takes a `protocol` argument rather than importing the `robot`, `instruments`, and `labware` modules. A similar level of structural change would require a major version 3. Another major version bump would be if all of the default units switched to nanoliters instead of microliters (we won't do this, it's just an example). This major behavioral change would also require a major version 3.

The minor version of the API is increased whenever we add new functionality that might change the way a protocol is written, or when we want to make a behavior change to an aspect of the API but not the whole thing. For instance, if we added support for a new module, added an option for specifying volume units in the `aspirate` and `dispense` functions, or added a way to queue up actions from multiple different modules and execute them at the same time, we would increase the minor version of the API. Another minor version bump would be if we added automatic liquid level tracking, and the position at which the OT-2 aspirated from wells was now dynamic - some people might not want that change appearing suddenly in their well-tested protocol, so we would increase the minor version.

Expressing Versions

You must specify the API version you are targeting at the top of your Python protocol. This is done in the `metadata` block, using the key `'apiLevel'`:

```
from opentrons import protocol_api

metadata = {
    'apiLevel': '2.7',
    'author': 'A. Biologist'}

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

This key exists alongside the other elements of the metadata.

Version specification is required by the system. If you do not specify your target API version, you will not be able to simulate or run your protocol.

The version you specify determines the features and behaviors available to your protocol. For instance, if Opentrons adds the ability to set the volume units in a call to `aspirate` in version 2.1, then you must specify version 2.1 in your metadata. A protocol like this:

```
from opentrons import protocol_api

metadata = {
    'apiLevel': '2.0',
    'author': 'A. Biologist'}

def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', '1')
    plate = protocol.load_labware('corning_96_wellplate_380ul', '2')
    left = protocol.load_instrument('p300_single', 'left', tip_racks=[tiprack])

    left.pick_up_tip()
    left.aspirate(volume=50000, location=plate['A1'], units='nanoliters')
```

would cause an error, because the `units` argument is not present in API version 2.0. This protects you from accidentally using features not present in your specified API version, and keeps your protocol portable between API versions.

In general, you should closely consider what features you need in your protocol, and keep your specified API level as low as possible. This makes your protocol work on a wider range of OT-2 software versions.

Determining What Version Is Available

Since version 3.15.0 of the OT-2 software and Opentrons App, the maximum supported API level of your OT-2 is visible in the Information card in the Opentrons App for your OT-2.

This maximum supported API level is the highest API level you can specify in a protocol. If you upload a protocol that specifies a higher API level than the OT-2 software supports, the OT-2 cannot simulate or run your protocol.

Determining What Features Are In What Version

As you read the documentation on this site, you will notice that all documentation on features, function calls, available properties, and everything else about the Python Protocol API notes which API version it was introduced in. Keep this information in mind when specifying your protocol's API version. The version statement will look like this:

New in version 2.0.

API and OT-2 Software Versions

This table lists the correspondence between Protocol API versions and robot software versions.

API Version	Introduced In OT-2 Software
1.0	3.0.0
2.0	3.14.0
2.1	3.15.2
2.2	3.16.0
2.3	3.17.0
2.4	3.17.1
2.5	3.19.0
2.6	3.20.0
2.7	3.21.0

Changes in API Versions

Version 2.1

- You can now specify a label when loading labware into a module with the `label` parameter of `ModuleContext.load_labware()` just like you can when loading labware into your protocol with *`ProtocolContext.load_labware()`* (page 60).

Version 2.2

- You should now specify magnetic module engage height using the `height_from_base` parameter, which specifies the height of the top of the magnet from the base of the labware. For more, see *Engage* (page 21).
- Return tip will now use pre-defined heights from hardware testing. For more information, see *Return Tip* (page 36).
- When using the return tip function, tips are no longer added back into the tip tracker. For more information, see *Return Tip* (page 36).

Version 2.3

- Magnetic Modules GEN2 and Temperature Modules GEN2 are now supported; you can load them with the names `"magnetic module gen2"` and `"temperature module gen2"`, respectively
- All pipettes will return tips to tipracks from a higher position to avoid possible collisions
- During a *Mix* (page 40), the pipette will no longer move up to clear the liquid in between every dispense and following aspirate
- You can now access the temperature module's status via the `status` property of `` `ModuleContext.TemperatureModuleContext` ``

Version 2.4

- **The following improvements were made to the *touch_tip* command:**
  - The speed for *touch_tip* can now be lowered down to 1 mm/s
  - *touch_tip* no longer moves diagonally from the X direction -> Y direction
  - Takes into account geometry of the deck and modules

Version 2.5

- **New *Utility Commands* (page 41) were added:**
  - *ProtocolContext.set_rail_lights()* (page 63): turns robot rail lights on or off
  - *ProtocolContext.rail_lights_on* (page 63): describes whether or not the rail lights are on
  - *ProtocolContext.door_closed* (page 60): describes whether the robot door is closed

Version 2.6

- GEN2 Single pipettes now default to flow rates equivalent to 10 mm/s plunger speeds
  - Protocols that manually configure pipette flow rates will be unaffected
  - For a comparison between API Versions, see *Defaults* (page 32)

Version 2.7

- You can now move both pipettes simultaneously on the robot! See *InstrumentContext.pair_with()* (page 69) for further information on how to use this new feature.

  This feature is still under development.
- Calling *InstrumentContext.has_tip()* (page 67) will return whether a particular instrument has a tip attached or not.

5.3 Labware

When writing a protocol, you must inform the Protocol API about the labware you will be placing on the OT-2's deck.

When you load labware, you specify the name of the labware (e.g. `'corning_96_wellplate_360ul_flat'`), and the slot on the OT-2's deck in which it will be placed (e.g. `'2'`). The first place to look for the names of labware should always be the Opentrons Labware Library[28], where Opentrons maintains a database of labware, their names in the API, 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[29]) and `'opentrons_96_tiprack_300ul'` (the Opentrons standard 300 µL tiprack[30]).

In the example given in the *Overview* (page 2) section, we loaded labware like this:

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

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

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

```
plate = protocol.load_labware('corning_96_wellplate_360ul_flat',
                              location='2',
                              label='any-name-you-want')
```

Labware is loaded into a protocol using *ProtocolContext.load_labware()* (page 60), which returns *opentrons.protocol_api.labware.Labware* (page 73) object.

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

Finding Labware

Default Labware

The OT-2 has a set of labware well-supported by Opentrons defined internally. This set of labware is always available to protocols. This labware can be found on the Opentrons Labware Library[31]. You can copy the load names that should be passed to `protocol.load_labware` statements to get the correct definitions.

Custom Labware

If you have a piece of labware that is not in the Labware Library, you can create your own definition using the Opentrons Labware Creator[32]. Before using the Labware Creator, you should read the introduction article here[33].

Once you have created your labware and saved it as a `.json` file, you can add it to the Opentrons App by clicking "More" and then "Labware". Once you have added your labware to the Opentrons App, it will be available to all Python Protocol API version 2 protocols uploaded to your robot through that Opentrons App. If other people will be using this custom labware definition, they must also add it to their Opentrons App. You can find a support article about this custom labware process here[34].

Accessing Wells in Labware

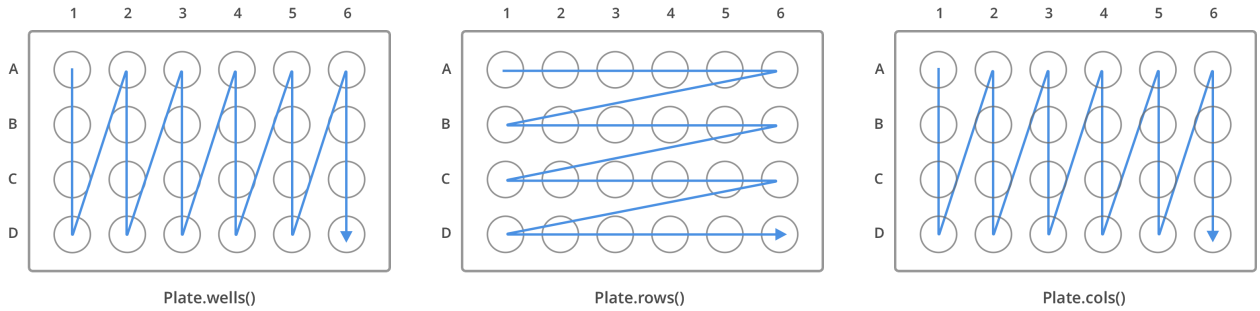
Well Ordering

When writing a protocol, 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
'''
metadata = {'apiLevel': '2.7'}

def run(protocol):
```

(continues on next page)

[31] https://labware.opentrons.com
[32] https://labware.opentrons.com/create/
[33] https://support.opentrons.com/en/articles/3136504-creating-custom-labware-definitions
[34] https://support.opentrons.com/en/articles/3136506-using-labware-in-your-protocols

(continued from previous page)

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

New in version 2.0.

Accessor Methods

There are many different ways to access wells inside labware. Different methods are useful in different contexts. The table below lists out the methods available to access wells and their differences.

Method Name	Returns
Labware.wells() (page 76)	List of all wells, i.e. `[labware:A1, labware:B1, labware:C1...]`
Labware.rows() (page 75)	List of a list ordered by row, i.e `[[labware:A1, labware:A2...], [labware:B1, labware:B2..]]`
Labware.columns() (page 73)	List of a list ordered by column, i.e. `[[labware:A1, labware:B1..], [labware:A2, labware:B2..]]`
Labware.wells_by_name() (page 76)	Dictionary with well names as keys, i.e. `{'A1': labware:A1, 'B1': labware:B1}`
Labware.rows_by_name() (page 75)	Dictionary with row names as keys, i.e. `{'A': [labware:A1, labware:A2..], 'B': [labware:B1, labware:B2]}`
Labware.columns_by_name() (page 74)	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 76):

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

New in version 2.0.

List Access From `wells`

Wells can be referenced by their 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[35]). Whichever well access method you use, your protocol will be most maintainable if you use only one access method consistently.

New in version 2.0.

Accessing Groups of Wells

When describing a liquid transfer, you can point to groups of wells for the liquid's source and/or destination. Or, you can get a group of wells and loop (or iterate) through them.

You can access a specific row or column of wells by using the *Labware.rows_by_name()* (page 75) and *Labware.columns_by_name()* (page 74) methods on a labware. These methods both return a dictionary with the row or column name as the keys:

```
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
```

Since these methods return either lists or dictionaries, you can iterate through them as you would regular Python data structures.

For example, to access the individual wells of row `'A'` in a well plate, you can 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.

New in version 2.0.

Specifying Position Within Wells

The functions listed above (in the *Accessing Wells in Labware* (page 13) 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 76) objects. *Well* (page 76) objects have some useful methods on them, which allow you to more closely specify the location to which the OT-2 should move *inside* a given well.

Each of these methods returns an object called a *opentrons.types.Location* (page 87), which encapsulates a position in deck coordinates (see *Deck Coordinates* (page 93)) and a well with which it is associated. This lets

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

you further manipulate the positions returned by these methods. All *InstrumentContext* (page 63) methods that involve positions accept these *Location* (page 87) objects.

Position Modifiers

Top

The method *Well.top()* (page 77) returns a position at the top center of the well. This is a good position to use for *Blow Out* (page 40) or any other operation where you don't want to be contacting the liquid. In addition, *Well.top()* (page 77) 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
```

New in version 2.0.

Bottom

The method *Well.bottom()* (page 77) returns a position at the bottom center of the well. This is a good position to start when considering where to aspirate, or any other operation where you want to be contacting the liquid. In addition, *Well.bottom()* (page 77) 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 77) 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 pipette that collided will be too high in `z` until the next time it picks up a tip.

Note: If you are using this to change the position at which the robot does *Aspirate* (page 38) or *Dispense* (page 39) throughout the protocol, consider setting the default aspirate or dispense offset with *InstrumentContext.well_bottom_clearance* (page 73) (see *Default Positions Within Wells* (page 31)).

New in version 2.0.

Center

The method *Well.center()* (page 77) 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
```

New in version 2.0.

Manipulating Positions

The objects returned by the position modifier functions are all instances of *opentrons.types.Location* (page 87), which are named tuples[36] representing the combination of a point in space (another named tuple) and a reference to the associated *Well* (page 76) (or *Labware* (page 73), or slot name, depending on context).

To adjust the position within a well, you can use *Location.move()* (page 88). Pass it a *opentrons.types.Point* (page 88) representing a 3-dimensional offset. It will return a new location, representing the original location with that offset applied.

For example:

```
from opentrons import types

metadata = {'apiLevel': '2.7'}

def run(protocol):
    plate = protocol.load_labware(
    'corning_24_wellplate_3.4ml_flat', slot='1')

    # Get the center of well A1.
    center_location = plate['A1'].center()

    # Get a location 1 mm right, 1 mm back, and 1 mm up from the center of well A1.
    adjusted_location = center_location.move(types.Point(x=1, y=1, z=1))

    # Move to 1 mm right, 1 mm back, and 1 mm up from the center of well A1.
    pipette.move_to(adjusted_location)
```

New in version 2.0.

5.4 Hardware Modules

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

We currently support the Temperature, Magnetic and Thermocycler Modules.

Module Setup

Loading Your Module Onto the Deck

Like labware and pipettes, you must inform the Protocol API of the modules you will use in your protocol.

Use *ProtocolContext.load_module()* (page 61) to load a module. It will return an object representing the module.

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
     # Load a Magnetic Module GEN2 in deck slot 1.
     magnetic_module = protocol.load_module('magnetic module gen2', 1)
```

(continues on next page)

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

(continued from previous page)

```
    # Load a Temperature Module GEN1 in deck slot 3.
    temperature_module = protocol.load_module('temperature module', 3)
```

Note: When you load a module in a protocol, you inform the OT-2 that you want the specified module to be present. Even if you do not use the module anywhere else in your protocol, the Opentrons App and the OT-2 will not let your protocol proceed until all modules loaded with `load_module` are attached to the OT-2.

New in version 2.0.

Available Modules

The first parameter to *`ProtocolContext.load_module()`* (page 61), the module's *load name,* specifies the kind of module to load. Check the table below for the proper load name to use for each kind of module.

Some modules were added to the Protocol API later than others. Make sure you use a *Protocol API version* (page 9) high enough to support all the modules you want to use.

Module		Load name	Minimum *API version* (page 9)
Temperature Module	GEN1	`'temperature module'` or `'tempdeck'`	2.0
	GEN2	`'temperature module gen2'`	2.3
Magnetic Module	GEN1	`'magnetic module'` or `'magdeck'`	2.0
	GEN2	`'magnetic module gen2'`	2.3
Thermocycler Module		`'thermocycler module'` or `'thermocycler'`	2.0

GEN1 vs. GEN2 Modules

GEN2 modules are newer. They have improvements that make them more reliable and easier to use.

Identifying a GEN2 Module

You can determine if your module is a GEN2 model by inspecting the sides of the device for a label that specifies *GEN2*.

Changes with the GEN2 Temperature Module

The GEN2 Temperature Module has a plastic insulating rim around the plate, and plastic insulating shrouds designed to fit over our aluminum blocks. This mitigates an issue where the GEN1 Temperature Module would have trouble cooling to very low temperatures, especially if it shared the deck with a running Thermocycler.

Changes with the GEN2 Magnetic Module

The GEN2 Magnetic Module uses smaller magnets than the GEN1 version. This mitigates an issue where beads would be attracted even when the magnets were retracted.

This means it will take longer for the GEN2 module to attract beads.

Recommended Magnetic Module GEN2 bead attraction time:

- Total liquid volume <= 50 uL: 5 minutes
- Total liquid volume > 50 uL: 7 minutes

Loading Labware Onto Your Module

Like specifying labware that will be present on the deck of the OT-2, you must specify labware that will be present on the module you have just loaded. You do this using `ModuleContext.load_labware()`. For instance, to load a Temperature Module and specify an aluminum block for 2 mL tubes[37], you would do:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    module = protocol.load_module('Temperature Module', slot)
    my_labware = module.load_
↪labware('opentrons_24_aluminumblock_generic_2ml_screwcap',
                                      label='Temperature-Controlled Tubes')
```

Notice that when you load labware on a module, you don't specify the labware's deck slot. The labware is loaded on the module, on whichever deck slot the module occupies.

New in version 2.0.

Module and Labware Compatibility

It's up to you to make sure that the labware and module you chose make sense together. The Protocol API won't stop you from making nonsensical combinations, like a tube rack on a Thermocycler.

See: What labware can I use with my modules?[38]

Loading Custom Labware Into Your Module

Any custom labware added to your Opentrons App (see *Custom Labware* (page 13)) is accessible when loading labware onto a module.

New in version 2.1.

Note: In API version 2.0, `ModuleContext.load_labware()` only took a `load_name` argument. In API version 2.1 (introduced in Robot Software version 3.15.2) or higher you can now specify a label, version, and namespace (though most of the time you won't have to).

Using a Temperature Module

The Temperature Module acts as both a cooling and heating device. It can control the temperature of its deck between 4 °C and 95 °C with a resolution of 1 °C.

[37] https://labware.opentrons.com/opentrons_24_aluminumblock_generic_2ml_screwcap?category=aluminumBlock
[38] https://support.opentrons.com/en/articles/3540964-what-labware-can-i-use-with-my-modules

Temperature Modules are represented in code by *TemperatureModuleContext* (page 80) objects.

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

metadata = {'apiLevel': '2.7'}

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
```

New in version 2.0.

Set Temperature

To set the Temperature Module to 4 °C do the following:

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

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

Note: This is unlike version 1 of the Python API, in which you would have to use the separate function `wait_for_temperature` to block protocol execution until the Temperature Module was ready.

New in version 2.0.

Read the Current Temperature

You can read the current real-time temperature of the Temperature Module using the *TemperatureModuleContext.temperature* (page 82) property:

```
temp_mod.temperature
```

New in version 2.0.

Read the Target Temperature

You can read the current target temperature of the Temperature Module using the *TemperatureModuleContext.target* (page 82) property:

```
temp_mod.target
```

New in version 2.0.

Check the Status

The *TemperatureModuleContext.status* (page 82) property is a string that is one of `'heating'`, `'cooling'`, `'holding at target'` or `'idle'`.

```
temp_mod.status
```

Deactivate

This function will stop heating or cooling and will turn off the fan on the Temperature Module.

```
temp_mod.deactivate()
```

Note: You can also deactivate your temperature module through the Opentrons 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 when the protocol ends, is cancelled, or is reset.

After deactivating your Temperature module, you can later call *`TemperatureModuleContext.set_temperature()`* (page 82) to heat or cool phase again.

New in version 2.0.

Using a Magnetic Module

The Magnetic Module controls a set of permanent magnets which can move vertically. When the magnets are raised or engaged, they induce a magnetic field in the labware on the module. When they are lowered or disengaged, they do not.

The Magnetic Module is represented by a *`MagneticModuleContext`* (page 82) object.

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

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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
```

New in version 2.0.

Engage

The *`MagneticModuleContext.engage()`* (page 83) function raises the magnets to induce a magnetic field in the labware on top of the Magnetic Module. The height of the magnets can be specified in several different ways, based on internally stored default heights for labware:

- If neither `height_from_base`, `height` nor `offset` is specified **and** the labware is supported on the Magnetic Module, the magnets will raise to a reasonable default height based on the specified labware.

  ```
  mag_mod.engage()
  ```

  New in version 2.0.

- The recommended way to specify the magnets' position is to utilize the `height_from_base` parameter, which allows you to raise the height of the magnets relative to the base of the labware.

```
mag_mod.engage(height_from_base=13.5)
```

A `mag_mod.engage(height_from_base=0)` call should move the tops of the magnets to level with base of the labware.

New in version 2.2.

Note: There is a +/- 1 mmm variance across magnetic module units, using `height_from_base=0` might not be able to get the magnets to completely flush with base of the labware. Please test before carrying out your experiment to ensure the desired engage height for your labware.

- You can also specify `height`, which should be a distance in mm from the home position of the magnets.

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

New in version 2.0.

- An `offset` can be applied to move the magnets relatively from the default engage height of the labware, **if** the labware is supported on the Magnetic Module.

```
mag_mod.engage(offset=-2)
```

New in version 2.0.

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_from_labware` (since version 2.2) or `height`, must be specified.

New in version 2.0.

Disengage

```
mag_mod.disengage()
```

The Magnetic Module will disengage when the device is turned on. It will not auto-disengage otherwise unless you call *MagneticModuleContext.disengage()* (page 83) in your protocol.

New in version 2.0.

Check the Status

The *MagneticModuleContext.status* (page 84) property is a string that is one of `'engaged'` or `'disengaged'`.

```
mag_mod.status
```

Using a Thermocycler Module

The Thermocycler Module allows users to perform complete experiments that require temperature sensitive reactions such as PCR.

There are two heating mechanisms in the Thermocycler. One is the block in which samples are located; the other is the lid heating pad.

The block can control its temperature between 4 °C and 99 °C to the nearest 1 °C.

The lid can control its temperature between 37 °C to 110 °C. Please see our support article[39] on controlling the Thermocycler in the Opentrons App.

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

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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, it is not necessary to specify a slot. This is because the Thermocycler has a default position that covers Slots 7, 8, 10, and 11. This is the only valid location for the Thermocycler on the OT-2 deck.

New in version 2.0.

Lid Motor Control

The Thermocycler can control its temperature with the lid open or closed. When the lid of the Thermocycler is open, the pipettes can access the loaded labware. You can control the lid position with the methods below.

Open Lid

```
tc_mod.open_lid()
```

New in version 2.0.

Close Lid

```
tc_mod.close_lid()
```

New in version 2.0.

Lid Temperature Control

You can control when a lid temperature is set. It is recommended that you set the lid temperature before executing a Thermocycler profile (see *Thermocycler Profiles* (page 25)). The range of the Thermocycler lid is 37 °C to 110 °C.

[39] https://support.opentrons.com/en/articles/3469797-thermocycler-module

Set Lid Temperature

ThermocyclerContext.set_lid_temperature() (page 87) takes one parameter: 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)
```

New in version 2.0.

Block Temperature Control

To set the block temperature inside the Thermocycler, you can use the method *ThermocyclerContext.set_block_temperature()* (page 87). It takes five parameters: `temperature`, `hold_time_seconds`, `hold_time_minutes`, `ramp_rate` and `block_max_volume`. Only `temperature` is required; the two `hold_time` parameters, `ramp_rate`, and `block_max_volume` are optional.

Temperature

If you only specify a `temperature` in °C, the Thermocycler will hold this temperature indefinitely until powered off.

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

New in version 2.0.

Hold Time

If you set a `temperature` and a `hold_time`, the Thermocycler will hold the temperature for the specified amount of time. Time can be passed in as minutes or seconds.

With a hold time, it is important to also include the `block_max_volume` parameter. This is to ensure that the sample reaches the target temperature before the hold time counts down.

In the example below, the Thermocycler will hold the 50 µl samples at 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, block_max_
↪volume=50)
```

New in version 2.0.

Block Max Volume

The Thermocycler's block temperature controller varies its behavior based on the amount of liquid in the wells of its labware. Specifying an accurate volume allows the Thermocycler to precisely track the temperature of the samples. The `block_max_volume` parameter is specified in µL and is the volume of the most-full well in the labware that is loaded on the Thermocycler's block. If not specified, it defaults to 25 µL.

```
tc_mod.set_block_temperature(4, hold_time_seconds=20, block_max_volume=80)
```

New in version 2.0.

Ramp Rate

Lastly, you can modify the `ramp_rate` in °C/sec for a given `temperature`.

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

Warning: Do not modify the `ramp_rate` unless you know what you're doing.

New in version 2.0.

Thermocycler Profiles

The Thermocycler can rapidly cycle through temperatures to execute heat-sensitive reactions. These cycles are defined as profiles.

Thermocycler profiles are defined for the Protocol API as lists of dicts. Each dict should have a `temperature` key, which specifies the temperature of a profile step, and either or both of `hold_time_seconds` or `hold_time_minutes`, which specify the duration of the step. For instance, this profile commands the Thermocycler to drive its temperature to 10 °C for 30 seconds, and then 60 °C for 45 seconds:

```
profile = [
  {'temperature': 10, 'hold_time_seconds': 30},
  {'temperature': 60, 'hold_time_seconds': 45}]
```

Once you have written your profile, you command the Thermocycler to execute it using *`ThermocyclerContext.execute_profile()`* (page 85). This function executes your profile steps multiple times depending on the `repetitions` parameter. It also takes a `block_max_volume` parameter, which is the same as that of the *`ThermocyclerContext.set_block_temperature()`* (page 87) function.

For instance, you can execute the profile defined above 100 times for a 30 µL-per-well volume like this:

```
profile = [
  {'temperature': 10, 'hold_time_seconds': 30},
  {'temperature': 60, 'hold_time_seconds': 30}]

tc_mod.execute_profile(steps=profile, repetitions=100, block_max_volume=30)
```

Note: Temperature profiles only control the temperature of the *block* in the Thermocycler. You should set a lid temperature before executing the profile using *`ThermocyclerContext.set_lid_temperature()`* (page 87).

New in version 2.0.

Thermocycler Status

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

Basic Status

The `ThermocyclerContext.status` property is one of the strings `'holding at target'`, `'cooling'`, `'heating'`, or `'idle'`.

```
tc_mod.status
```

New in version 2.0.

Lid Position

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

```
tc_mod.lid_position
```

New in version 2.0.

Heated Lid Temperature Status

The current status of the heated lid temperature controller. It can be one of the strings `'holding at target'`, `'heating'`, `'idle'`, or `'error'`.

```
tc_mod.lid_temperature_status
```

New in version 2.0.

Block Temperature Status

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

```
tc_mod.block_temperature_status
```

New in version 2.0.

Thermocycler Deactivate

At some points in your protocol, you may want to deactivate specific temperature controllers of your Thermocycler. This can be done with three methods, *ThermocyclerContext.deactivate()* (page 85), *ThermocyclerContext.deactivate_lid()* (page 85), *ThermocyclerContext.deactivate_block()* (page 85).

Deactivate

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

```
tc_mod.deactivate()
```

Deactivate Lid

This deactivates only the heated lid of the Thermocycler.

```
tc_mod.deactivate_lid()
```

New in version 2.0.

Deactivate Block

This deactivates only the well block of the Thermocycler.

```
tc_mod.deactivate_block()
```

New in version 2.0.

5.5 Pipettes

When writing a protocol, you must inform the Protocol API about the pipettes you will be using on your OT-2. The Protocol API then creates software objects called *InstrumentContext* (page 63), that represent the attached pipettes.

Pipettes are loaded into a specific mount (`'left'` or `'right'`) on the OT-2 using the function *ProtocolContext.load_instrument()* (page 60) from the *ProtocolContext* (page 58) class. This will return an *InstrumentContext* (page 63) object. See *Building Block Commands* (page 35) and *Complex Commands* (page 45) for liquid handling commands from the *InstrumentContext* (page 63) class.

Loading A Pipette

Pipettes are specified in a protocol using the method *ProtocolContext.load_instrument()* (page 60). 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

metadata = {'apiLevel': '2.7'}

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])
```

New in version 2.0.

Note: When you load a pipette in a protocol, you inform the OT-2 that you want the specified pipette to be present. Even if you do not use the pipette anywhere else in your protocol, the Opentrons App and the OT-2 will not let your protocol proceed until all pipettes loaded with `load_instrument` are attached to the OT-2.

Multi-Channel Pipettes

All building block and advanced commands work with both single-channel (like `'p20_single_gen2'`) and multi-channel (like `'p20_multi_gen2'`) pipettes. To keep the interface to the Opentrons API consistent between single and multi-channel pipettes, commands treat the *backmost channel* (furthest from the door) of a multi-channel pipette as the location of the pipette. Location arguments to building block and advanced commands are specified for the backmost channel. This also means that offset changes (such as *Well.top()* (page 77) or *Well.bottom()* (page 77)) can be applied to the single specified well, and each channels of the pipette will be at the same position relative to the well that it is over.

Because there is only one motor in a multi-channel pipette, multi-channel pipettes will always aspirate and dispense on all channels simultaneously.

For instance, to aspirate from the first column of a 96-well plate you would write:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    # Load a tiprack for 300uL tips
    tiprack1 = protocol.load_labware('opentrons_96_tiprack_300ul', 1)
    # Load a wellplate
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat')

    # Load a P300 Multi GEN2 on the right mount
    right = protocol.load_instrument(
        'p300_multi_gen2', 'right', tip_rack=tiprack1)

    # Specify well A1 for pick_up_tip. The backmost channel of the
    # pipette moves to A1, which means the rest of the wells are above the
    # rest of the wells in column 1.
    right.pick_up_tip(tiprack1['A1'])

    # Similarly, specifying well A2 for aspirate means the pipette will
    # position its backmost channel over well A2, and the rest of the
    # pipette channels are over the rest of the wells of column 1
    right.aspirate(300, plate['A2'])

    # Dispense into column 3 of the plate with all 8 channels of the
    # pipette at the top of their respective wells
    right.dispense(300, plate['A3'].top())
```

In general, you should specify wells in the first row of a labware when you are using multi-channel pipettes. One common exception to this rule is when using 384-well plates. The spacing between the wells in a 384-well plate and the space between the nozzles of a multi-channel pipette means that a multi-channel pipette accesses every other well in a column. Specifying well A1 acesses every other well starting with the first (rows A, C, E, G, I, K, M, and O); specifying well B1 similarly accesses every other well, but starting with the second (rows B, D, F, H, J, L, N, and P).

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    # Load a tiprack for 300uL tips
    tiprack1 = protocol.load_labware('opentrons_96_tiprack_300ul', 1)
    # Load a wellplate
    plate = protocol.load_labware('corning_384_wellplate_112ul_flat')

    # Load a P300 Multi GEN2 on the right mount
    right = protocol.load_instrument(
        'p300_multi_gen2', 'right', tip_rack=tiprack1)

    # pick up a tip in preparation for aspiration
    right.pick_up_tip()

    # Aspirate from wells A1, C1, E1, G1, I1, K1, M1, and O1
    right.aspirate(300, plate['A1'])
```

(continues on next page)

(continued from previous page)

```
    # Dispense in wells B1, D1, F1, H1, J1, L1, N1, and P1
    right.dispense(300, plate['B1'])
```

This pattern of access applies to both building block commands and advanced commands.

Pipette Models

This table lists the model names, which are passed to *ProtocolContext.load_instrument()* (page 60), for each model of pipette sold by Opentrons.

Pipette Type	Model Name
P20 Single GEN2 (1 - 20 µL)	'p20_single_gen2'
P300 Single GEN2 (20 - 300 µL)	'p300_single_gen2'
P1000 Single GEN2 (100 - 1000 µL)	'p1000_single_gen2'
P300 Multi GEN2 (20-300 µL)	'p300_multi_gen2'
P20 Multi GEN2 (1-20 µL)	'p20_multi_gen2'
P10 Single (1 - 10 µL)	'p10_single'
P10 Multi (1 - 10 µL)	'p10_multi'
P50 Single (5 - 50 µL)	'p50_single'
P50 Multi (5 - 50 µL)	'p50_multi'
P300 Single (30 - 300 µL)	'p300_single'
P300 Multi (30 - 300 µL)	'p300_multi'
P1000 Single (100 - 1000 µL)	'p1000_single'

GEN2 Pipette Backward Compatibility

GEN2 pipettes have different volume ranges than GEN1 pipettes. However, each GEN2 pipette covers one or two GEN1 pipette volume ranges. For instance, with a range of 1 - 20 µL, the P20 Single GEN2 covers the P10 Single GEN1 (1 - 10 µL). If your protocol specifies a GEN1 pipette but you have a GEN2 pipette attached to your OT-2 with a compatible volume range, you can still run your protocol. The OT-2 will consider the GEN2 pipette to have the same minimum volume as the GEN1 pipette, so any advanced commands have the same behavior as before.

Specifically, the P20 GEN2s (single and multi) cover the entire P10 GEN1 range; the P300 Single GEN2 covers the entire P300 Single GEN1 range; and the P1000 Single GEN2 covers the entire P1000 Single GEN1 range.

If you have a P50 Single specified in your protocol, there is no automatic backward compatibility. If you want to use a GEN2 Pipette, you must change your protocol to load either a P300 Single GEN2 (if you are using volumes between 20 and 50 µL) or a P20 Single GEN2 (if you are using volumes below 20 µL).

If your protocol specifies a pipette and you attach a compatible pipette, the protocol will run, and the pipette will act the same as the pipette specified in your protocol - altering parameters like its minimum volume if necessary.

For instance, if your protocol specifies a P300 Multi, and you connect a P300 Multi GEN2, the pipette will act like a P300 Multi - it will set its minimum volume to 30 µL.

Adding Tip Racks

When you load a pipette, you can optionally specify a list of tip racks you will use to supply the pipette. This is done with the optional parameter `tip_racks` to *ProtocolContext.load_instrument()* (page 60). 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. This removes the need to specify tip locations in *InstrumentContext.pick_up_tip()* (page 69).

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

This is further discussed in *Building Block Commands* (page 35) and *Complex Commands* (page 45).

New in version 2.0.

Modifying Pipette Behaviors

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

Plunger Flow Rates

Opentrons pipettes aspirate or dispense at different rates. These flow rates can be changed on a loaded *InstrumentContext* (page 63) at any time, in units of µL/sec by altering *InstrumentContext.flow_rate* (page 67). This has the following attributes:

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

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

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware('opentrons_96_tiprack_300ul', '1')
    pipette = protocol.load_instrument(
        'p300_single', 'right', tip_racks=[tiprack])
    plate = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    pipette.pick_up_tip()

    # Aspirate at the default flowrate of 150 ul/s
    pipette.aspirate(50, plate['A1'])
    # Dispense at the default flowrate of 300 ul/s
    pipette.dispense(50, plate['A1'])

    # Change default aspirate speed to 50ul/s, 1/3 of the default
    pipette.flow_rate.aspirate = 50
    # this aspirate will be at 50ul/s
    pipette.aspirate(50, plate['A1'])
    # this dispense will be the default 300 ul/s
    pipette.dispense(50, plate['A1'])

    # Slow down dispense too
    pipette.flow_rate.dispense = 50
    # This is still at 50 ul/s
    pipette.aspirate(50, plate['A1'])
    # This is now at 50 ul/s as well
    pipette.dispense(50, plate['A1'])

    # Also slow down the blow out flowrate from its default
    pipette.flow_rate.blow_out = 100
    pipette.aspirate(50, plate['A1'])
```

(continues on next page)

(continued from previous page)

```
    # This will be much slower
    pipette.blow_out()

    pipette.drop_tip()
```

InstrumentContext.speed (page 70) 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.

New in version 2.0.

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 geometries, liquids, or experimental protocols. While you can specify the exact location within a well in direct calls to *InstrumentContext.aspirate()* (page 64) and *InstrumentContext.dispense()* (page 65) (see the *Specifying Position Within Wells* (page 15) section), you cannot use this method in complex commands like *InstrumentContext.transfer()* (page 71), and it can be cumbersome to specify the position every time.

Instead, you can use the attribute *InstrumentContext.well_bottom_clearance* (page 73) 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

metadata = {'apiLevel': '2.7'}

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'])
```

New in version 2.0.

Gantry 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 require 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 65). 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

metadata = {'apiLevel': '2.7'}

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)))
```

New in version 2.0.

Per-Axis Speed Limits

In addition to controlling the overall speed of motions, you can set per-axis speed limits for the OT-2's axes. Unlike the overall speed, which is controlled per-instrument, axis speed limits take effect for both pipettes and all motions. These can be set for the `X` (left-and-right gantry motion), `Y` (forward-and-back gantry motion), `Z` (left pipette up-and-down motion), and `A` (right pipette up-and-down motion) using *ProtocolContext.max_speeds* (page 62). This works like a dictionary, where the keys are axes, assigning to a key sets a max speed, and deleting a key or setting it to `None` resets that axis's limit to the default:

```
metadata = {'apiLevel': '2.7'}

def run(protocol):
    protocol.max_speeds['X'] = 50  # limit x axis to 50 mm/s
    del protocol.max_speeds['X']  # reset x axis limit
    protocol.max_speeds['A'] = 10  # limit a axis to 10 mm/s
    protocol.max_speeds['A'] = None  # reset a axis limit
```

You cannot set limits for the pipette plunger axes with this mechanism; instead, set the flow rates or plunger speeds as described in *Plunger Flow Rates* (page 30).

New in version 2.0.

Defaults

Head Speed: 400 mm/s

Well Bottom Clearances

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

p20_single_gen2

- **Aspirate Default:**
  - On API Version 2.5 and previous: 3.78 µL/s
  - On API Version 2.6 and subsequent: 7.56 µL/s
- **Dispense Default:**
  - On API Version 2.5 and previous: 3.78 µL/s
  - On API Version 2.6 and subsequent: 7.56 µL/s
- **Blow Out Default:**
  - On API Version 2.5 and previous: 3.78 µL/s
  - On API Version 2.6 and subsequent: 7.56 µL/s
- Minimum Volume: 1 µL
- Maximum Volume: 20 µL

p300_single_gen2

- **Aspirate Default:**
  - On API Version 2.5 and previous: 46.43 µL/s
  - On API Version 2.6 and subsequent: 92.86 µL/s
- **Dispense Default:**
  - On API Version 2.5 and previous: 46.43 µL/s
  - On API Version 2.6 and subsequent: 92.86 µL/s
- **Blow Out Default:**
  - On API Version 2.5 and previous: 46.43 µL/s
  - On API Version 2.6 and subsequent: 92.86 µL/s
- Minimum Volume: 20 µL
- Maximum Volume: 300 µL

p1000_single_gen2

- **Aspirate Default:**
  - On API Version 2.5 and previous: 137.35 µL/s
  - On API Version 2.6 and subsequent: 274.7 µL/s
- **Dispense Default:**
  - On API Version 2.5 and previous: 137.35 µL/s
  - On API Version 2.6 and subsequent: 274.7 µL/s
- **Blow Out Default:**
  - On API Version 2.5 and previous: 137.35 µL/s
  - On API Version 2.6 and subsequent: 274.7 µL/s
- Minimum Volume: 100 µL
- Maximum Volume: 1000 µL

p20_multi_gen2

- Aspirate Default: 7.6 µL/s
- Dispense Default: 7.6 µL/s
- Blow Out Default: 7.6 µL/s
- Minimum Volume: 1 µL
- Maximum Volume: 20 µL

p300_multi_gen2

- Aspirate Default: 94 µL/s
- Dispense Default: 94 µL/s
- Blow Out Default: 94 µL/s
- Minimum Volume: 20 µL
- Maximum Volume: 300 µL

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

5.6 Building Block Commands

Building block, or basic, commands are the smallest individual actions that can be completed on an OT-2. For example, the complex command `transfer` (see *Complex Commands* (page 45)) executes a series of `pick_up_tip()`, `aspirate()`, `dispense()` and `drop_tip()` basic commands.

The examples in this section would be added to the following:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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_gen2', mount='left')
    # the example code below would go here, inside the run function
```

This loads a Corning 96 Well Plate[40] in slot 2 and a Opentrons 300 µL Tiprack[41] in slot 3, and uses a P300 Single GEN2 pipette.

Tip Handling

When the OT-2 handle liquids with, it constantly exchanges old, used tips for new ones to prevent cross-contamination between wells. Tip handling uses the functions *`InstrumentContext.pick_up_tip()`* (page 69), *`InstrumentContext.drop_tip()`* (page 66), and *`InstrumentContext.return_tip()`* (page 70).

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

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 69) will move the pipette over to the specified tip, then press down into it to create a vacuum seal. The below example picks up the tip at location `'A1'` of the tiprack previously loaded in slot 3.

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

If you have associated a tiprack with your pipette such as in the *Pipettes* (page 27) or *Protocols and Instruments* (page 58) sections, then you can simply call

```
pipette.pick_up_tip()
```

This will use the next available tip from the list of tipracks passed in to the `tip_racks` argument of *ProtocolContext.load_instrument()* (page 60).

New in version 2.0.

Drop Tip

Once finished with a tip, the pipette will remove the tip when we call *InstrumentContext.drop_tip()* (page 66). You can specify where to drop the tip by passing in a location. The below example drops the tip back at its original location on the tip rack. If no location is specified, the OT-2 will drop the tip in the fixed trash in slot 12 of the deck.

```
pipette.pick_up_tip()
pipette.drop_tip(tiprack['A1'])  # drop back in A1 of the tiprack
pipette.pick_up_tip()
pipette.drop_tip()  # drop in the fixed trash on the deck
```

New in version 2.0.

Return Tip

To return the tip to the original location, you can call *InstrumentContext.return_tip()* (page 70). The example below will automatically return the tip to `'A3'` on the tip rack.

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

In API version 2.2 or above:

```
tip_rack = protocol.load_labware(
        'opentrons_96_tiprack_300ul', 1)
pipette = protocol.load_instrument(
    'p300_single_gen2', mount='left', tip_racks=[tip_rack])

pipette.pick_up_tip() # picks up tip_rack:A1
pipette.return_tip()
pipette.pick_up_tip() # picks up tip_rack:B1
```

In API version 2.0 and 2.1:

```
tip_rack = protocol.load_labware(
        'opentrons_96_tiprack_300ul', 1)
pipette = protocol.load_instrument(
    'p300_single_gen2', mount='left', tip_racks=[tip_rack])

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

Iterating Through Tips

For this section, instead of using the protocol defined above, consider this setup:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        'corning_96_wellplate_360ul_flat', 2)
    tip_rack_1 = protocol.load_labware(
        'opentrons_96_tiprack_300ul', 3)
    tip_rack_2 = protocol.load_labware(
        'opentrons_96_tiprack_300ul', 4)
    pipette = protocol.load_instrument(
        'p300_single_gen2', mount='left', tip_racks=[tip_rack_1, tip_rack_2])
```

This loads a Corning 96 Well Plate[42] in slot 2 and two Opentrons 300ul Tiprack[43] in slots 3 and 4 respectively, and uses a P300 Single GEN2 pipette.

When a list of tip racks is associated with a pipette in its `tip_racks` argument, the pipette will automatically pick up the next unused tip in the list whenever you call *InstrumentContext.pick_up_tip()* (page 69). The pipette will first use all tips in the first tiprack, then move on to the second, and so on:

```
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 you try to *InstrumentContext.pick_up_tip()* (page 69) again when all the tips have been used, the Protocol API will show you an error:

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

To change the location of the first tip used by the pipette, you can use *InstrumentContext.starting_tip* (page 71):

[42] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[43] https://labware.opentrons.com/opentrons_96_tiprack_300ul

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

To reset the tip tracking, you can call *InstrumentContext.reset_tipracks()* (page 70):

```
# Use up all tips
for _ in range(96+96):
     pipette.pick_up_tip()
     pipette.return_tip()

# Reset the tip tracker
pipette.reset_tipracks()

# Picks up a tip from well A1 of the first tip rack
pipette.pick_up_tip()
```

New in version 2.0.

To check whether you should pick up a tip or not, you can utilize *InstrumentContext.has_tip()* (page 67):

```
for block in range(3):
    if block == 0 and not pipette.has_tip:
        pipette.pick_up_tip()
    else:
        m300.mix(mix_repetitions, 250, d)
        m300.blow_out(s.bottom(10))
        m300.return_tip()
```

New in version 2.7.

Liquid Control

This section describes the *InstrumentContext* (page 63) 's liquid-handling commands.

The examples in this section should be inserted in the following:

```
metadata = {'apiLevel': '2.7'}

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_gen2', mount='left', tip_
↪racks=[tiprack])
    pipette.pick_up_tip()
    # example code goes here
```

This loads a Corning 96 Well Plate[44] in slot 2 and a Opentrons 300ul Tiprack[45] in slot 3, and uses a P300 Single GEN2 pipette.

Aspirate

To aspirate is to pull liquid up into the pipette's tip. When calling *InstrumentContext.aspirate()* (page 64) on a pipette, you can specify the volume to aspirate in µL, where to aspirate from, and how fast to aspirate liquid.

[44] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[45] https://labware.opentrons.com/opentrons_96_tiprack_300ul

```
pipette.aspirate(50, plate['A1'], rate=2.0)  # aspirate 50uL from plate:A1
```

Now the pipette's tip is holding 50 µL.

The `location` parameter is either a well (like `plate['A1']`) or a position within a well, like the return value of `plate['A1'].bottom`.

The `rate` parameter is a multiplication factor of the pipette's default aspiration flow rate. The default aspiration flow rate for all pipettes is in the *Defaults* (page 32) section.

You can also simply specify the volume to aspirate, and not mention a location. The pipette 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 100 µL.

Note: In version 1 of this API, `aspirate` (and `dispense`) would inspect the types of the `volume` and `location` arguments and do the right thing if you specified only a location or specified location and volume out of order. In this and future versions of the Python Protocol API, this is no longer true. Like any other Python function, if you are specifying arguments by position without using their names, you must always specify them in order.

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 77) (see *Specifying Position Within Wells* (page 15)) 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 73) (see *Default Positions Within Wells* (page 31)).

New in version 2.0.

Dispense

To dispense is to push out liquid from the pipette's tip. The usage of *`InstrumentContext.dispense()`* (page 65) in the Protocol API is similar to *`InstrumentContext.aspirate()`* (page 64), in that you can specify volume in µL and location, or only volume.

```
pipette.dispense(50, plate['B1'], rate=2.0) # dispense 50uL to plate:B1 at twice the␣
↪normal rate
pipette.dispense(50)                # dispense 50uL to current position at the normal␣
↪rate
```

The `location` parameter is either a well (like `plate['A1']`) or a position within a well, like the return value of `plate['A1'].bottom`.

The `rate` parameter is a multiplication factor of the pipette's default dispense flow rate. The default dispense flow rate for all pipettes is in the *Defaults* (page 32) section.

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 77) (see *Specifying Position Within Wells* (page 15)) 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 73) (see *Default Positions Within Wells* (page 31)).

Note: In version 1 of this API, `dispense` (and `aspirate`) would inspect the types of the `volume` and `location` arguments and do the right thing if you specified only a location or specified location and volume out of order. In this and future versions of the Python Protocol API, this is no longer true. Like any other Python function, if you are specifying arguments by position without using their names, you must always specify them in order.

New in version 2.0.

Blow Out

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

When calling *InstrumentContext.blow_out()* (page 65), you can 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
```

New in version 2.0.

Touch Tip

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

When calling *InstrumentContext.touch_tip()* (page 71) on a pipette, you have the option to specify a location where the tip will touch the inner walls.

InstrumentContext.touch_tip() (page 71) 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'], speed=100) # touch tip within plate:B1 at 100 mm/s
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)
```

New in version 2.0.

Mix

To mix is to perform a series of `aspirate` and `dispense` commands in a row on a single location. Instead of having to write those commands out every time, you can call *InstrumentContext.mix()* (page 68).

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

```
# mix 4 times, 100uL, in plate:A2
pipette.mix(4, 100, plate['A2'])
# mix 3 times, 50uL, in current location
pipette.mix(3, 50)
```

(continues on next page)

(continued from previous page)

```
# mix 2 times, pipette's max volume, in current location
pipette.mix(2)
```

Note: In API Versions 2.2 and earlier, mixes consist of aspirates and then immediate dispenses. In between these actions, the pipette moves up and out of the target well. In API Version 2.3 and later, the pipette will not move between actions.

New in version 2.0.

Air Gap

When dealing with certain liquids, you may need to aspirate air after aspirating the liquid to prevent it from sliding out of the pipette's tip. A call to *InstrumentContext.air_gap()* (page 64) with a volume in µL will aspirate that much air into the tip. `air_gap` takes up to two arguments: `air_gap(volume, height)`:

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

New in version 2.0.

Utility Commands

Move To

You can use *InstrumentContext.move_to()* (page 68) to move a pipette to any location on the deck.

For example, you can move to the first tip in your tip rack:

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

Unlike commands that require labware, like *Aspirate* (page 38) or *Dispense* (page 39), *InstrumentContext.move_to()* (page 68) deals with *types.Location* (page 87) instances, which combine positions in *Deck Coordinates* (page 93) and associated *Labware* (page 73) instances. You don't have to create them yourself; this is what is returned from methods such as *Well.top()* (page 77) and *Well.bottom()* (page 77). It does mean, however, that you can't move to a well directly; you must use *Well.top()* (page 77) or build a *types.Location* (page 87) yourself.

You can also specify at what height you would like the robot to move to inside of a location using *Well.top()* (page 77) and *Well.bottom()* (page 77) methods on that location (more on these methods and others like them in the *Specifying Position Within Wells* (page 15) 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'].top(), 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'].top())
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'].top())
```

New in version 2.0.

Delay

Sometimes you need to pause your protocol, for instance to wait for something to incubate. You can use *ProtocolContext.delay()* (page 59) to pause your protocol for a specific amount of time. `delay` is a method of *ProtocolContext* (page 58) since it concerns the protocol and the OT-2 as a whole.

The value passed into `delay()` is the number of minutes or seconds the OT-2 will wait until moving on to the next command.

```
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 63) will pause protocol execution at a specific step. You can resume by pressing 'resume' in your Opentrons App. You can optionally specify a message that will be displayed in the Opentrons App when protocol execution pauses.

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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

    # The OT-2 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')
```

New in version 2.0.

Homing

You can manually request that the OT-2 home during protocol execution. This is typically not necessary; however, 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 OT-2, you can call *ProtocolContext.home()* (page 60).

To home a specific pipette's Z axis and plunger, you can call *InstrumentContext.home()* (page 67).

To home a specific pipette's plunger only, you can call *InstrumentContext.home_plunger()* (page 67).

None of these functions take any arguments:

```
from opentrons import protocol_api, types

metadata = {'apiLevel': '2.7'}

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
```

New in version 2.0.

Comment

The method *ProtocolContext.comment()* (page 59) lets you display messages in the Opentrons App during protocol execution:

```
from opentrons import protocol_api, types

metadata = {'apiLevel': '2.7'}

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

New in version 2.0.

Control and Monitor Robot Rail Lights

You can turn the robot rail lights on or off in the protocol using *ProtocolContext.set_rail_lights()* (page 63):

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

def run(protocol: protocol_api.ProtocolContext):
    # turn on robot rail lights
    protocol.set_rail_lights(True)

    # turn off robot rail lights
    protocol.set_rail_lights(False)
```

New in version 2.5.

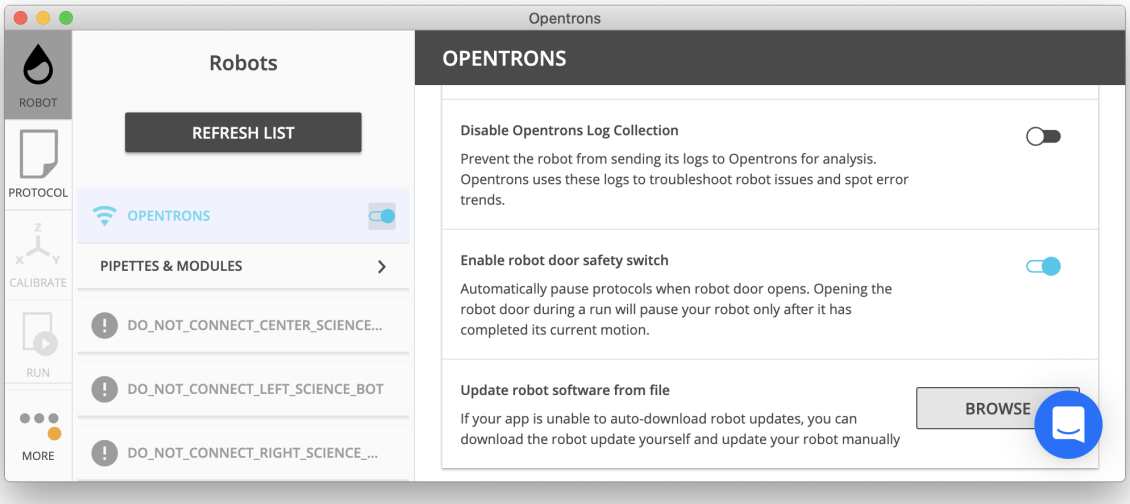
You can also check whether the rail lights are on or off in the protocol using *ProtocolContext.rail_lights_on* (page 63):

```
protocol.rail_lights_on  # returns True when the lights are on,
                         # False when the lights are off
```

New in version 2.5.

Monitor Robot Door

The door safety switch feature flag has been added to the OT-2 software since the 3.19.0 release. Enabling the feature flag allows your robot to pause a running protocol and prohibit the protocol from running when the robot door is open.



You can also check whether or not the robot door is closed at a specific point in time in the protocol using *ProtocolContext.door_closed* (page 60):

```
protocol.door_closed  # return True when the door is closed,
                      # False when the door is open
```

Note: Both the top window and the front door must be closed in order for the robot to report the door is closed.

Warning: If you chose to enable the door safety switch feature flag, you should only use *ProtocolContext.door_closed* (page 60) as a form of status check, and should not use it to control robot behavior. If you wish to implement custom method to pause or resume protocol using *ProtocolContext.door_closed* (page 60), make sure you have first disabled the feature flag.

New in version 2.5.

5.7 Complex Commands

Overview

The commands in this section execute long or complex series of the commands described in the *Building Block Commands* (page 35) section. These advanced commands make it easier to handle larger groups of wells and repetitive actions.

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

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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)
    tiprack_multi = protocol.load_labware('opentrons_96_tiprack_300ul', 3)
    pipette = protocol.load_instrument('p300_single', mount='left', tip_
↪racks=[tiprack])
    pipette_multi = protocol.load_instrument('p300_multi', mount='right', tip_
↪racks=[tiprack_multi])

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

This loads a Corning 96 Well Plate[46] in slot 1 and a Opentrons 300 µL Tiprack[47] in slot 2 and 3, and uses a P300 Single pipette and a P300 Multi pipette.

You can follow along and simulate the protocol using our protocol simulator, which can be installed by following the instructions at *Using Python For Protocols* (page 5).

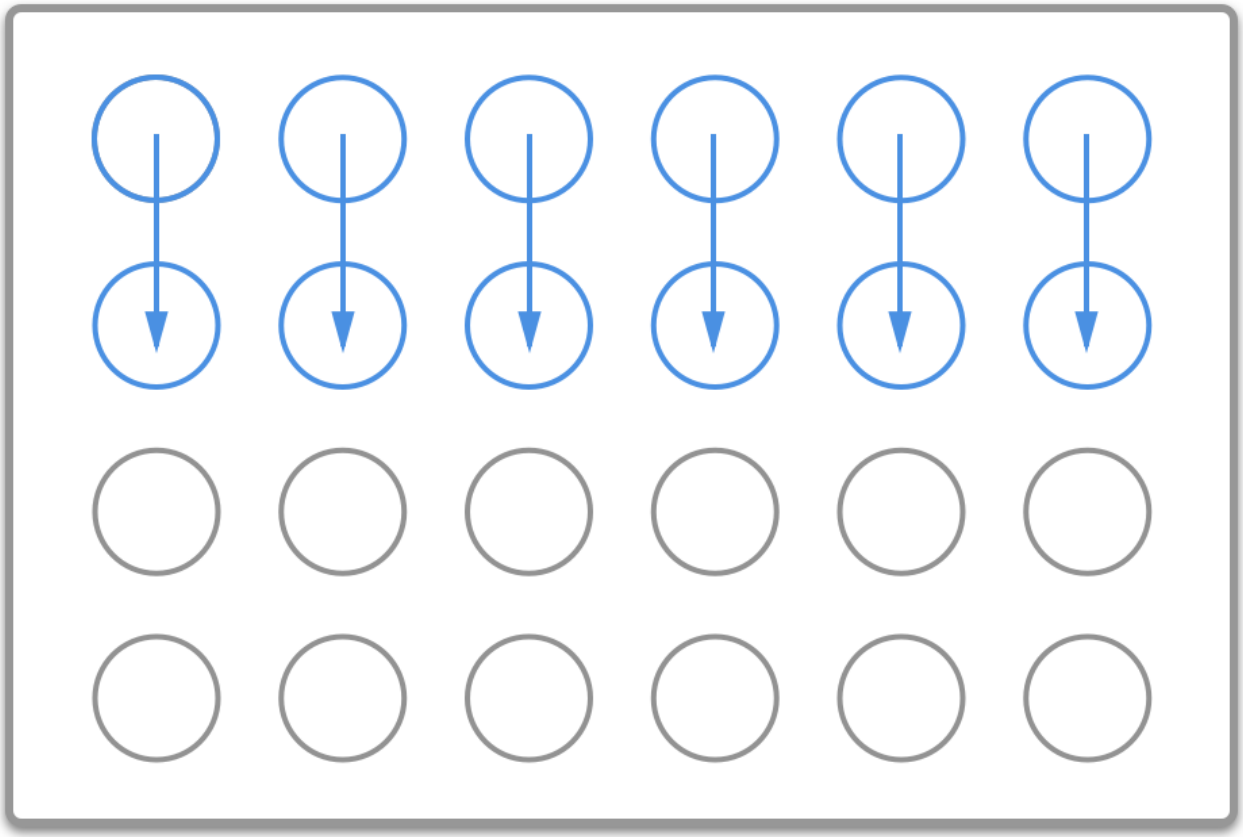
There are three complex liquid handling commands:

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 71)	Yes	Yes	Yes
InstrumentContext.distribute() (page 66)	Yes	Yes	No
InstrumentContext.consolidate() (page 65)	No	Yes	Yes

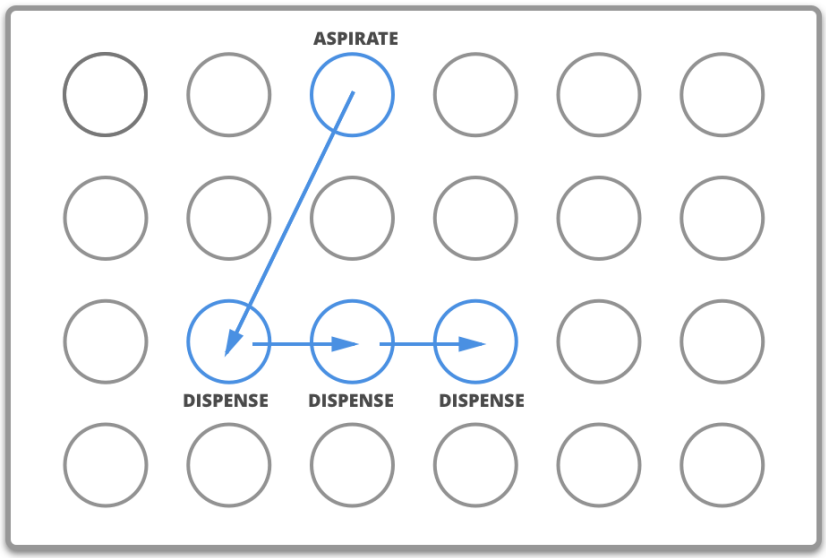
You can also refer to these images for further clarification.

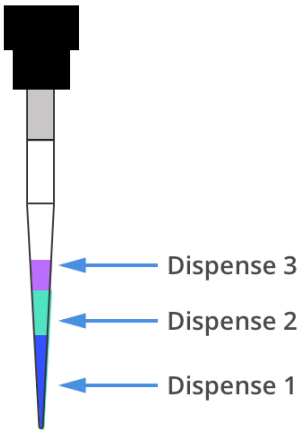
[46] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[47] https://labware.opentrons.com/opentrons_96_tiprack_300ul

Transfer

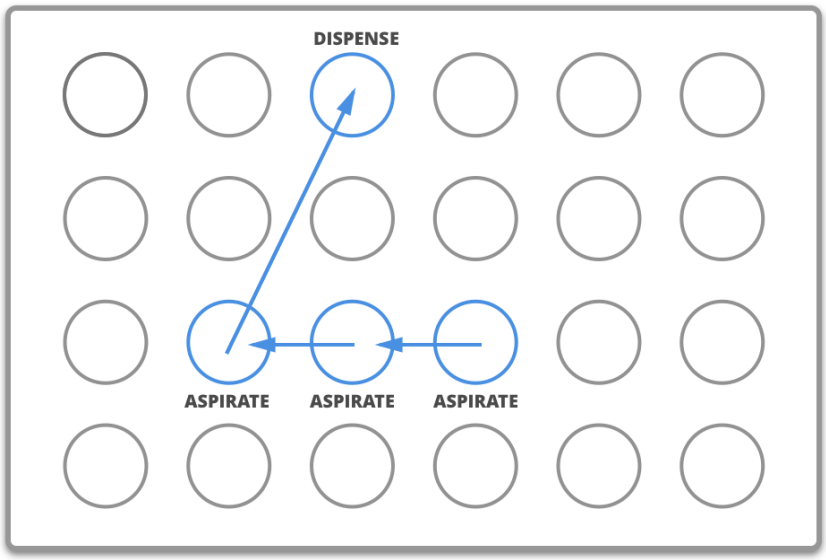


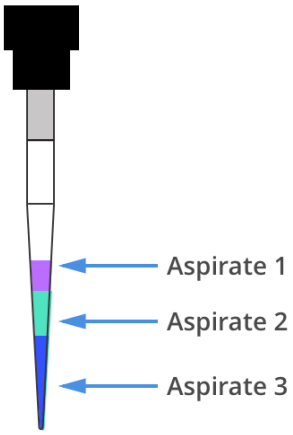
Distribute





Consolidate





Parameters

Parameters for the complex liquid handling are listed here in order of operation. Check out the *Complex Liquid Handling Parameters* (page 53) 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 µL	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 µL to hold in tip while dispensing; better accuracies in multi-dispense	0	10% of pipette max volume	0

Transfer

The most versatile complex liquid handling function is *InstrumentContext.transfer()* (page 71). For a majority of use cases you will most likely want to use this complex command.

Below you will find a few scenarios using the *InstrumentContext.transfer()* (page 71) command.

New in version 2.0.

Basic

This example below transfers 100 µL from well `'A1'` to well `'B1'` using the P300 Single pipette, 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'])
```

When you are using a multi-channel pipette, you can transfer the entire column (8 wells) in the plate to another using:

```
pipette.transfer(100, plate.wells_by_name()['A1'], plate.wells_by_name()['A2'])
```

Note: In API Versions 2.0 and 2.1, multichannel pipettes could only access the first row of a 384 well plate, and access to the second row would be ignored. If you need to transfer from all wells of a 384-well plate, please make sure to use API Version 2.2

Note: Multichannel pipettes can only access a limited number of rows in a plate during *transfer*, *distribute* and *consolidate*: the first row (wells A1 - A12) of a 96-well plate, and (since API Version 2.2) the first two rows (wells A1 - B24) for a 384-well plate. Wells specified outside of the limit will be ignored.

Transfer commands will automatically create entire series of *InstrumentContext.aspirate()* (page 64), *InstrumentContext.dispense()* (page 65), and other *InstrumentContext* (page 63) commands.

Large Volumes

Volumes larger than the pipette's `max_volume` (see *Defaults* (page 32)) 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. This will be a one to one transfer from where well `A1`'s contents are transferred to well `A2`, well `B1`'s contents to `B2`,and so on. This is the scenario displayed in the *Transfer* (page 46) 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"
```

New in version 2.0.

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
```

(continues on next page)

(continued from previous page)

```
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"
```

New in version 2.0.

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"
```

New in version 2.0.

Distribute and Consolidate

InstrumentContext.distribute() (page 66) and *InstrumentContext.consolidate()* (page 65) are similar to *InstrumentContext.transfer()* (page 71), but optimized for specific uses. *InstrumentContext.distribute()* (page 66) is optimized for taking a large volume from a single (or a small number) of source wells, and distributing it to many smaller volumes in destination wells. Rather than using one-to-one transfers, it dispense many times for each aspirate. *InstrumentContext.consolidate()* (page 65) is optimized for taking small volumes from many source wells and consolidating them into one (or a small number) of destination wells, aspirating many times for each dispense.

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. This is the scenario described by the *Consolidate* (page 47) graphic.

```
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 not combine the transfers - it will aspirate from one source, dispense into the target, then aspirate from the other source.

```
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"
```

New in version 2.0.

Distribute

Volumes from the same source well are combined within the same tip, so that one aspirate can provide for multiple dispenses. This is the scenario in the *Distribute* (page 46) graphic.

```
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"
```

(continues on next page)

(continued from previous page)

```
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"
```

The pipette will aspirate more liquid than it intends to dispense by the minimum volume of the pipette. This is called the `disposal_volume`, and can be specified in the call to `distribute`.

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"
```

New in version 2.0.

Order of Operations In Complex Commands

Parameters to complex commands add behaviors to the generated complex command in a specific order which cannot be changed. Specifically, advanced commands execute their atomic commands in this order:

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, if any
9. Blow Out

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

Complex Liquid Handling Parameters

Below are some examples of the parameters described in the *Parameters* (page 47) table.

new_tip

This parameter handles tip logic. You have options of the strings `'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'`.

New in version 2.0.

Always Get a New Tip

Transfer commands will by default use the same 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"
```

(continues on next page)

(continued from previous page)

```
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"
```

Use One Tip

The default behavior of complex commands is to use one 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='once')     # use one tip (default behavior)
```

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"
```

(continues on next page)

(continued from previous page)

```
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, compelx commands 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"
```

New in version 2.0.

touch_tip

A *Touch Tip* (page 40) 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"
```

New in version 2.0.

blow_out

A *Blow Out* (page 40) 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"
```

New in version 2.0.

mix_before, mix_after

A *Mix* (page 40) can be performed before every aspirate by setting `mix_before=`, and after every dispense by setting `mix_after=`. The value of `mix_before=` or `mix_after=` must be a tuple; the first value is the number of repetitions, the second 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"
```

New in version 2.0.

air_gap

An *Air Gap* (page 41) can be performed after every aspirate by setting `air_gap=volume`, where the value is the volume of air in µL to aspirate after aspirating the liquid. The entire volume in the tip, air gap and the liquid volume,

will be dispensed all at once at the destination specified in the complex command.

```
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 1.0 speed
Air gap
Aspirating 20 uL from well A1 in "1" at 1.0 speed
Dispensing 120.0 uL into well A2 in "1"
Dropping tip well A1 in "12"
```

New in version 2.0.

disposal_volume

When dispensing multiple times from the same tip in *InstrumentContext.distribute()* (page 66), it is recommended to aspirate an extra amount of liquid to be disposed of after distributing. This added `disposal_volume` can be set as an optional argument.

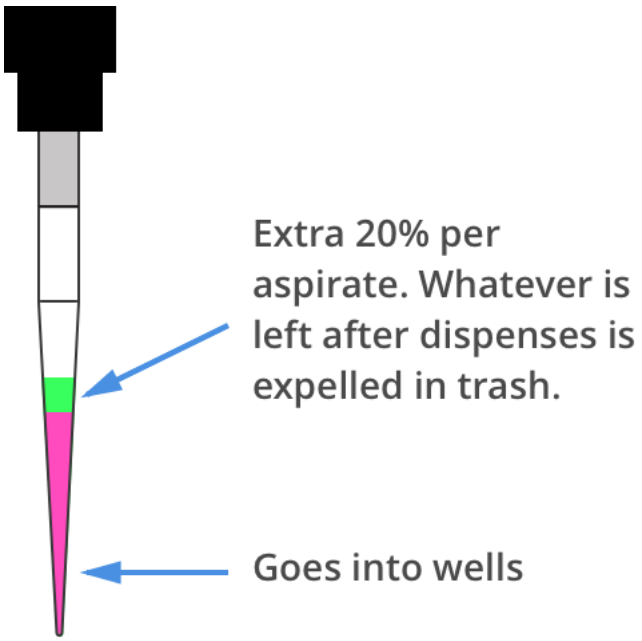
The default disposal volume is the pipette's minimum volume (see *Defaults* (page 32)), which will be blown out at the trash after the dispenses.

```
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,



New in version 2.0.

5.8 API Version 2 Reference

Protocols and Instruments

class opentrons.protocol_api.contexts.**ProtocolContext**(*loop: asyncio.events.AbstractEventLoop = None, hardware: Union[opentrons.hardware_control.thread_manager.ThreadManager, opentrons.hardware_control.adapters.SynchronousAdapter, HasLoop] = None, broker=None, bundled_labware: Dict[str, LabwareDefinition] = None, bundled_data: Dict[str, bytes] = None, extra_labware: Dict[str, LabwareDefinition] = None, api_version: opentrons.protocols.types.APIVersion = 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 `legacy_api.api.robot`, which is provided only for back compatibility and should be used less and less as time goes by.

New in version 2.0.

property api_version

Return the API version supported by this protocol context.

The supported API version was specified when the protocol context was initialized. It may be lower than the highest version supported by the robot software. For the highest version supported by the robot software, see `protocol_api.MAX_SUPPORTED_VERSION`.

New in version 2.0.

classmethod build_using(*protocol: Union[opentrons.protocols.types.JsonProtocol, opentrons.protocols.types.PythonProtocol], *args, **kwargs*)
Build an API instance for the specified parsed protocol

This is used internally to provision the context with bundle contents or api levels.

property 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.

New in version 2.0.

cleanup(*self*)
Finalize and clean up the protocol context.

clear_commands(*self*)
New in version 2.0.

commands(*self*)
New in version 2.0.

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

The value of the message is computed during protocol simulation, so cannot be used to communicate real-time information from the robot's actual run.

New in version 2.0.

connect(*self*, *hardware: opentrons.hardware_control.api.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 58); *disconnect()* (page 60) simply creates a new simulator and replaces the current hardware with it.

New in version 2.0.

property 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 62) to see a dict of labwares and *loaded_modules* (page 62) to see a dict of modules. For advanced control you can delete an item of labware from the deck with e.g. `del protocol.deck['1']` to free a slot for new labware. (Note that for each slot only the last labware used in a command will be available for calibration in the OpenTrons UI, and that the tallest labware on the deck will be calculated using only currently loaded labware, meaning that the labware loaded should always reflect the labware physically on the deck (or be higher than the labware on the deck).

New in version 2.0.

delay(*self*, *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.

New in version 2.0.

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

New in version 2.0.

property door_closed
Returns True if the robot door is closed

New in version 2.5.

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

It has one well and should be accessed like labware in your protocol. e.g. `protocol.fixed_trash['A1']`

New in version 2.0.

home (*self*)
Homes the robot.

New in version 2.0.

is_simulating (*self*) → bool
New in version 2.0.

load_instrument (*self, instrument_name: str, mount: Union[opentrons.types.Mount, str], tip_racks: List[opentrons.protocol_api.labware.Labware] = None, replace: bool = False*) → '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 (page 88) *or str*) – The mount in which this instrument should be attached. This can either be an instance of the enum type *types.Mount* (page 88) or one of the strings *'left'* and *'right'*.
- **tip_racks** (List[*Labware* (page 73)]) – A list of tip racks from which to pick tips if *InstrumentContext.pick_up_tip()* (page 69) 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*.

New in version 2.0.

load_labware (*self, load_name: str, location: 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.

New in version 2.0.

load_labware_by_name(*self, load_name: str, location: Union[int, str], label: str = None, namespace: str = None, version: int = 1*) → opentrons.protocol_api.labware.Labware

New in version 2.0.

load_labware_from_definition(*self, labware_def: 'LabwareDefinition', location: 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'
- **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.

New in version 2.0.

load_module(*self, module_name: str, location: Union[int, str, NoneType] = None, configuration: str = None*) → 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 60) 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 62).

Parameters

- **module_name** (*str*) – The name or model 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.
- **configuration** – Used to specify the slot configuration of the Thermocycler. Only Valid in Python API Version 2.4 and later. If you wish to use the non-full plate configuration, you must pass in the key word value *semi*

Returns ModuleContext The loaded and initialized `ModuleContext`.

New in version 2.0.

property 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 60). 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 60), 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

New in version 2.0.

property 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 59).

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

New in version 2.0.

property 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 61). 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 61), 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.

New in version 2.0.

property 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
```

(continues on next page)

(continued from previous page)

```
                                    # 10 mm/s
    protocol.max_speeds['X'] = None  # reset to default
```

New in version 2.0.

pause (*self*, *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 63) is called.

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

New in version 2.0.

property rail_lights_on
Returns True if the rail lights are on

New in version 2.5.

resume (*self*)
Resume a previously-paused protocol

New in version 2.0.

set_rail_lights (*self*, *on: bool*)
Controls the robot rail lights

Parameters **on** (*bool*) – If true, turn on rail lights; otherwise, turn off.

New in version 2.5.

temp_connect (*self*, *hardware: opentrons.hardware_control.api.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
```

class opentrons.protocol_api.contexts.**InstrumentContext**(*ctx: ProtocolContext*, *hardware_mgr: HardwareManager*, *mount: types.Mount*, *log_parent: logging.Logger*, *at_version: APIVersion*, *tip_racks: List[Labware] = None*, *trash: Labware = None*, *default_speed: float = 400.0*, *requested_as: str = None*, ***config_kwargs*)
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 64) or *distribute()* (page 66)) are defined here for convenience.

In general, this class should not be instantiated directly; rather, instances are returned from *ProtocolContext.load_instrument()* (page 60).

New in version 2.0.

air_gap (*self*, *volume: 'float' = None*, *height: 'float' = None*) → '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 64), *dispense()* (page 65))

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.

New in version 2.0.

property api_version
New in version 2.0.

aspirate (*self*, *volume: 'float' = None*, *location: 'Union[types.Location*, *Well]' = None*, *rate: 'float' = 1.0*) → '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 64) 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 67).
- **location** – Where to aspirate from. If *location* is a *Well* (page 76), the robot will aspirate from `well_bottom_clearance.aspirate` mm above the bottom of the well. If *location* is a *Location* (page 87) (i.e. the result of *Well.top()* (page 77) or *Well.bottom()* (page 77)), 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'])`

New in version 2.0.

blow_out (*self*, *location: 'Union[types.Location, Well]' = None*) → 'InstrumentContext'
Blow liquid out of the tip.

If *dispense* (page 65) 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 76) or *Location* (page 87) 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 64), *dispense()* (page 65))

Returns This instance

New in version 2.0.

property channels
The number of channels on the pipette.

New in version 2.0.

consolidate (*self*, *volume: 'float'*, *source: 'List[Well]'*, *dest: 'Well'*, **args*, ***kwargs*) → '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 71). Some arguments are changed. Specifically, `mix_before`, if specified, is ignored and `disposal_volume` is ignored and set to 0.

Returns This instance

New in version 2.0.

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

New in version 2.0.

property 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 68).

New in version 2.0.

delay (*self*)
New in version 2.0.

dispense (*self*, *volume: 'float' = None*, *location: 'Union[types.Location, Well]' = None*, *rate: 'float' = 1.0*) → '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 65)).

Parameters

- **volume** (*int or float*) – The volume of liquid to dispense, in microliters. If not specified, defaults to *current_volume* (page 65).
- **location** – Where to dispense into. If *location* is a *Well* (page 76), the robot will dispense into `well_bottom_clearance.dispense` mm above the bottom of the well. If *location* is a *Location* (page 87) (i.e. the result of *Well.top()* (page 77) or *Well.bottom()* (page 77)), 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'])`

New in version 2.0.

distribute(*self*, *volume: 'float'*, *source: 'Well'*, *dest: 'List[Well]'*, **args*, ***kwargs*) → '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 71). Some arguments are changed. Specifically, `mix_after`, if specified, is ignored and `disposal_volume`, if not specified, is set to the minimum volume of the pipette

Returns This instance

New in version 2.0.

drop_tip(*self*, *location: 'Union[types.Location, Well]' = None*, *home_after: 'bool' = True*) → 'InstrumentContext'

Drop the current tip.

If no location is passed, the Pipette will drop the tip into its *trash_container* (page 72), 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 76). 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 76) 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 87); for instance, you can call *instr.drop_tip(tiprack.wells()[0].top())*.

Parameters

- **location** (*types.Location* (page 87) or *Well* (page 76) or None) – The location to drop the tip
- **home_after** – Whether to home the plunger after dropping the tip (defaults to `True`). The plungeer must home after dropping tips because the ejector shroud that pops the tip off the end of the pipette is driven by the plunger motor, and may skip steps when dropping the tip.

Returns This instance

New in version 2.0.

property 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 70); 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 70) uses the units of the linear speed of the plunger inside the pipette. Because *speed* (page 70) and *flow_rate* (page 67) 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
```

New in version 2.0.

property has_tip

Whether this instrument has a tip attached or not. :type: bool

New in version 2.7.

Type returns

home(*self*) → 'InstrumentContext'

Home the robot.

Returns This instance.

New in version 2.0.

home_plunger(*self*) → 'InstrumentContext'

Home the plunger associated with this mount

Returns This instance.

New in version 2.0.

property hw_pipette

View the information returned by the hardware API directly.

Raises a *types.PipetteNotAttachedError* (page 88) if the pipette is no longer attached (should not happen).

New in version 2.0.

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

New in version 2.0.

property min_volume
New in version 2.0.

mix(*self*, *repetitions: 'int' = 1*, *volume: 'float' = None*, *location: 'Union[types.Location, Well]' = None*, *rate: 'float' = 1.0*) → '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 67).

Parameters

- **repetitions** – how many times the pipette should mix (default: 1)
- **volume** – number of microliters to mix (default: *max_volume* (page 67))
- **location** (`types.Location` (page 87)) – 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.

New in version 2.0.

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

New in version 2.0.

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

New in version 2.0.

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

Parameters

- **location** (*types.Location* (page 87)) – 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 65). This controls the straight linear speed of the motion; to limit individual axis speeds, you can use `ProtocolContext.max_speeds`.

New in version 2.0.

property name

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

New in version 2.0.

pair_with (*self*, *instrument: 'InstrumentContext'*) → 'PairedInstrumentContext'

This function allows you to pair both of your pipettes and use them simultaneously. The function implicitly decides a primary and secondary pipette based on which instrument you call this function on.

Parameters **instrument** – The secondary instrument you wish to use

Raises **UnsupportedInstrumentPairingError** – If you try to pair

pipettes that are not currently supported together. :returns: PairedInstrumentContext: This is the object you will call commands on.

This function returns a `PairedInstrumentContext`. The building block commands are the same as an individual pipette's building block commands found at *Building Block Commands* (page 35), and when you want to move pipettes simultaneously you need to use the `PairedInstrumentContext`.

Limitations: 1. This function utilizes a "primary" and "secondary" pipette to make positional decisions. The consequence of doing this is that all X & Y positions are based on the primary pipette only. 2. At this time, only pipettes of the same type are supported for pipette pairing. This means that you cannot utilize a P1000 Single channel and a P300 Single channel at the same time.

```
from opentrons import protocol_api

# metadata
metadata = {
    'protocolName': 'My Protocol',
    'author': 'Name <email@address.com>',
    'description': 'Simple paired pipette protocol,
    'apiLevel': '2.7'
}

def run(ctx: protocol_api.ProtocolContext):
    right_pipette = ctx.load_instrument(
        'p300_single_gen2', 'right')
    left_pipette = ctx.load_instrument('p300_single_gen2', 'left')

    # In this scenario, the right pipette is the primary pipette
    # while the left pipette is the secondary pipette. All XY
    # locations will be based on the right pipette.
    right_paired_with_left = right_pipette.pair_with(left_pipette)
    right_paired_with_left.pick_up_tip()
    right_paired_with_left.drop_tip()

    # In this scenario, the left pipette is the primary pipette
    # while the right pipette is the secondary pipette. All XY
    # locations will be based on the left pipette.
    left_paired_with_right = left_pipette.pair_with(right_pipette)
    left_paired_with_right.pick_up_tip()
    left_paired_with_right.drop_tip()
```

Note: Before using this method, you should seriously consider whether this is the best fit for your use-case especially given the limitations listed above.

New in version 2.7.

pick_up_tip (*self*, *location: 'Union[types.Location, Well]' = None*, *presses: 'int' = None*, *increment: 'float' = None*) → '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 71) 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 76). 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 63).
- 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 87); for instance, you can call `instr.pick_up_tip(tiprack.wells()[0].top())`.

Parameters

- **location** (*types.Location* (page 87) or *Well* (page 76) 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

New in version 2.0.

reset_tipracks (*self*)

Reload all tips in each tip rack and reset starting tip

New in version 2.0.

property return_height

The height to return a tip to its tiprack.

New in version 2.2.

return_tip (*self*, *home_after: 'bool' = True*) → '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

New in version 2.0.

property 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 67); 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 67) uses the units of the volumetric flow rate of liquid into or out of the tip. Because *speed* (page 70) and *flow_rate* (page 67) 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
```

New in version 2.0.

property starting_tip

The starting tip from which the pipette pick up

New in version 2.0.

property 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 69) without arguments.

New in version 2.0.

touch_tip(*self*, *location: 'Well' = None*, *radius: 'float' = 1.0*, *v_offset: 'float' = -1.0*, *speed: 'float' = 60.0*) → 'InstrumentContext'

Touch the pipette tip to the sides of a well, with the intent of removing left-over droplets

Parameters

- **location** (*Well* (page 76) 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 64), *dispense()* (page 65))

Returns This instance

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

New in version 2.0.

transfer(*self*, *volume: 'Union[float, Sequence[float]]'*, *source: 'AdvancedLiquidHandling'*, *dest: 'AdvancedLiquidHandling'*, *trash=True*, ***kwargs*) → '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 63) commands, like *aspirate()* (page 64) and *dispense()* (page 65), 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 71) will occur following each *aspirate()* (page 64) and *dispense()* (page 65). If set to *False* (default behavior), no *touch_tip()* (page 71) will occur.
- *blow_out* (`boolean`) – If *True*, a *blow_out()* (page 65) will occur following each *dispense()* (page 65), but only if the pipette has no liquid left in it. If set to *False* (default), no *blow_out()* (page 65) will occur.
- *mix_before* (`tuple`) – The tuple, if specified, gives the amount of volume to *mix()* (page 68) preceding each *aspirate()* (page 64) 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 68) after each *dispense()* (page 65) during the transfer. The tuple is interpreted as (repetitions, volume).
- *disposal_volume* (`float`) – (*distribute()* (page 66) 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

New in version 2.0.

property 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 66) or *blow_out()* (page 65) without arguments.

New in version 2.0.

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

New in version 2.0.

property 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 64) or *dispense()* (page 65) is given a *Well* (page 76) rather than a full *Location* (page 87), 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
```

New in version 2.0.

Labware and Wells

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

This module provides things like *Labware* (page 73), and *Well* (page 76) 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: LabwareDefinition*, *parent: opentrons.types.Location*, *label: str = None*, *api_level: opentrons.protocols.types.APIVersion = None*)
This class represents a labware, such as a PCR plate, a tube rack, reservoir, 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 76), *wells()* (page 76), *columns()* (page 73), *rows()* (page 75), *rows_by_name()* (page 75), and *columns_by_name()* (page 74). 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]
```

property api_version
New in version 2.0.

property calibrated_offset
New in version 2.0.

columns (*self*, **args*) → List[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

New in version 2.0.

columns_by_index (*self*) → Dict[str, List[opentrons.protocol_api.labware.Well]]
New in version 2.0.

columns_by_name (*self*) → Dict[str, 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

New in version 2.0.

property 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.

New in version 2.0.

property is_tiprack
New in version 2.0.

property load_name
The API load name of the labware definition

New in version 2.0.

property magdeck_engage_height
New in version 2.0.

property name
Can either be the canonical name of the labware, which is used to load it, or the label of the labware specified by a user.

New in version 2.0.

next_tip (*self*, *num_tips: int = 1*, *starting_tip: opentrons.protocol_api.labware.Well = None*) → 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 76) meeting the target criteria, or None

property parameters
Internal properties of a labware including type and quirks

New in version 2.0.

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

New in version 2.0.

previous_tip (*self*, *num_tips: int = 1*) → 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 76) meeting the target criteria, or `None`

property quirks
Quirks specific to this labware.

New in version 2.0.

reset (*self*)
Reset all tips in a tiprack

New in version 2.0.

return_tips (*self*, *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 76), 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 75).

Parameters

- **start_well** (*Well* (page 76)) – The *Well* (page 76) into which to return a tip.
- **num_channels** (*int*) – The number of channels for the current pipette

rows (*self*, **args*) → List[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

New in version 2.0.

rows_by_index (*self*) → Dict[str, List[opentrons.protocol_api.labware.Well]]
New in version 2.0.

rows_by_name (*self*) → Dict[str, 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

New in version 2.0.

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

property tip_length
New in version 2.0.

property uri
A string fully identifying the labware.

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

New in version 2.0.

use_tips (*self*, *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 76)) – The *Well* (page 76) 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 (*self*, *idx*) → opentrons.protocol_api.labware.Well
Deprecated—use result of *wells* or *wells_by_name*

New in version 2.0.

wells (*self*, **args*) → 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

New in version 2.0.

wells_by_index (*self*) → Dict[str, opentrons.protocol_api.labware.Well]
New in version 2.0.

wells_by_name (*self*) → 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

New in version 2.0.

exception opentrons.protocol_api.labware.**OutOfTipsError**

exception opentrons.protocol_api.labware.**TipSelectionError**

class opentrons.protocol_api.labware.**Well**(*well_props: WellDefinition*, *parent: opentrons.types.Location*, *display_name: str*, *has_tip: bool*, *api_level: opentrons.protocols.types.APIVersion*, *name: str = None*)

The Well class represents a single well in a *Labware* (page 73)

It provides functions to return positions used in operations on the well such as *top()* (page 77), *bottom()* (page 77)

property api_version

New in version 2.0.

bottom(*self*, *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

New in version 2.0.

center(*self*) → 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))

New in version 2.0.

property diameter

New in version 2.0.

property has_tip

New in version 2.0.

property parent

New in version 2.0.

top(*self*, *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

New in version 2.0.

property well_name

New in version 2.7.

opentrons.protocol_api.labware.**get_all_labware_definitions**() → List[str]

Return a list of standard and custom labware definitions with load_name + name_space **+** version existing on the robot

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

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: Dict[str, ForwardRef('LabwareDefinition')] = None*, *extra_defs: Dict[str, ForwardRef('LabwareDefinition')] = None*, *api_level: opentrons.protocols.types.APIVersion = 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 87) 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.
- **api_level** (*APIVersion*) – the API version to set for the loaded labware instance. The *Labware* (page 73) will conform to this level. If not specified, defaults to MAX_SUPPORTED_VERSION.

opentrons.protocol_api.labware.**load_from_definition**(*definition: 'LabwareDefinition'*, *parent: opentrons.types.Location*, *label: str = None*, *api_level: opentrons.protocols.types.APIVersion = 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 87) 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
- **api_level** (*APIVersion*) – the API version to set for the loaded labware instance. The *Labware* (page 73) will conform to this level. If not specified, defaults to MAX_SUPPORTED_VERSION.

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

Walk the tree of wells and labwares and extract quirks

opentrons.protocol_api.labware.**save_definition**(*labware_def: 'LabwareDefinition'*, *force: bool = False*, *location: pathlib.Path = None*) → 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.**select_tiprack_from_list_paired_pipettes**(*tip_racks: List[opentrons.protocol_api.p_channels: int, s_channels: int, starting_point: opentrons.protocol_api.labware.* = *None*) → Tuple[opentrons.protocol_api.opentrons.protocol_api.labware.

Helper function utilized in `PairedInstrumentContext` to determine which pipette tiprack to pick up from.

If a starting point is specified, this method with check that the parent of that tip was correctly filtered.

If a starting point is not specified, this method will filter tipracks until it finds a well that is not empty.

Returns A Tuple of the tiprack and well to move to. In this

instance the starting well is specific to the primary pipette. :raises TipSelectionError: if the starting tip specified does not exist in the filtered tipracks.

opentrons.protocol_api.labware.**verify_definition**(*contents: Union[~AnyStr, ForwardRef('LabwareDefinition'), Dict[str, Any]]*) → 'LabwareDefinition'

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: ProtocolContext, hw_module: opentrons.hardware_control.modules.tempdec geometry: opentrons.protocols.geometry.module_geomet at_version: opentrons.protocols.types.APIVersion, loop: asyncio.events.AbstractEventLoop*)

An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 61) 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.

New in version 2.0.

property api_version
New in version 2.0.

await_temperature (*self*, *celsius: float*)
Wait until module reaches temperature, in C.

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

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

New in version 2.3.

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

New in version 2.0.

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

Returns ModuleGeometry

New in version 2.0.

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

New in version 2.0.

load_labware (*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters

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

Returns The initialized and loaded labware object.

New in version 2.1: The *label, namespace,* and *version* parameters.

New in version 2.0.

load_labware_by_name(*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: str = None*) → opentrons.protocol_api.labware.Labware

Specify the presence of a labware on the module, using an inline definition.

Parameters

- **definition** – The labware definition.
- **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.

Returns The initialized and loaded labware object.

New in version 2.0.

load_labware_object(*self*, *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()* (page 81).

Returns The properly-linked labware object

New in version 2.0.

set_temperature(*self*, *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

New in version 2.0.

property status

The status of the module.

Returns 'holding at target', 'cooling', 'heating', or 'idle'

New in version 2.3.

property target

Current target temperature in C

New in version 2.0.

property temperature

Current temperature in C

New in version 2.0.

class opentrons.protocol_api.contexts.**MagneticModuleContext**(*ctx: ProtocolContext*, *hw_module: opentrons.hardware_control.modules.magdeck.Ma*
geometry: opentrons.protocols.geometry.module_geometry.M
at_version: opentrons.protocols.types.APIVersion, *loop: asyncio.events.AbstractEventLoop*)

An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 61).

New in version 2.0.

property api_version
New in version 2.0.

calibrate(*self*)
Calibrate the Magnetic Module.

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

New in version 2.0.

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

New in version 2.0.

engage(*self*, *height: float = None*, *offset: float = None*, *height_from_base: 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`, `height_from_base` nor `offset` is specified, the magnets will raise to a reasonable default height based on the specified labware.
- The recommended way to adjust the height of the magnets is to specify `height_from_base`, which should be a distance in mm relative to the base of the labware that is on the magnetic module
- 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` or `height_from_labware` must be specified.

Parameters

- **height_from_base** – The height to raise the magnets to, in mm from the base of the labware
- **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

New in version 2.1: The *height_from_base* parameter.

New in version 2.0.

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

Returns ModuleGeometry

New in version 2.0.

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

New in version 2.0.

load_labware(*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters

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

Returns The initialized and loaded labware object.

New in version 2.1: The *label, namespace,* and *version* parameters.

New in version 2.0.

load_labware_by_name(*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware
New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: str = None*) → opentrons.protocol_api.labware.Labware
Specify the presence of a labware on the module, using an inline definition.

Parameters

- **definition** – The labware definition.
- **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.

Returns The initialized and loaded labware object.

New in version 2.0.

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

New in version 2.0.

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

New in version 2.0.

class opentrons.protocol_api.contexts.**ThermocyclerContext**(*ctx: ProtocolContext*, *hw_module: opentrons.hardware_control.modules.thermocycler.Th*
geometry: opentrons.protocols.geometry.module_geometry.Ther
at_version: opentrons.protocols.types.APIVersion, *loop: asyncio.events.AbstractEventLoop*)

An object representing a connected Temperature Module.

It should not be instantiated directly; instead, it should be created through *ProtocolContext.load_module()* (page 61).

New in version 2.0.

property api_version
New in version 2.0.

property block_target_temperature
Target temperature in degrees C

New in version 2.0.

property block_temperature
Current temperature in degrees C

New in version 2.0.

property block_temperature_status
New in version 2.0.

close_lid(*self*)
Closes the lid

New in version 2.0.

deactivate(*self*)
Turn off the well block temperature controller, and heated lid

New in version 2.0.

deactivate_block(*self*)
Turn off the well block temperature controller

New in version 2.0.

deactivate_lid(*self*)
Turn off the heated lid

New in version 2.0.

execute_profile(*self, steps: List[Dict[str, float]], repetitions: int, block_max_volume: float = None*)
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()* (page 87) method with keys 'temperature', 'hold_time_seconds', and 'hold_time_minutes'.
- **repetitions** – The number of times to repeat the cycled steps.
- **block_max_volume** – The maximum volume of any individual well of the loaded labware. If not supplied, the thermocycler will default to 25µL/well.

New in version 2.0.

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

Returns ModuleGeometry

New in version 2.0.

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

New in version 2.0.

property lid_position
Lid open/close status string

New in version 2.0.

property lid_target_temperature
Target temperature in degrees C

New in version 2.0.

property lid_temperature
Current temperature in degrees C

New in version 2.0.

property lid_temperature_status
New in version 2.0.

load_labware(*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware
Specify the presence of a piece of labware on the module.

Parameters

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

Returns The initialized and loaded labware object.

New in version 2.1: The *label, namespace,* and *version* parameters.

New in version 2.0.

load_labware_by_name(*self*, *name: str*, *label: str = None*, *namespace: str = None*, *version: int = 1*) → opentrons.protocol_api.labware.Labware
New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: str = None*) → opentrons.protocol_api.labware.Labware
Specify the presence of a labware on the module, using an inline definition.

Parameters

- **definition** – The labware definition.

- **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.

Returns The initialized and loaded labware object.

New in version 2.0.

load_labware_object(*self*, *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()* (page 86).

Returns The properly-linked labware object

New in version 2.0.

open_lid(*self*)

Opens the lid

New in version 2.0.

set_block_temperature(*self*, *temperature: float*, *hold_time_seconds: float = None*, *hold_time_minutes: float = None*, *ramp_rate: float = None*, *block_max_volume: float = None*)

Set the target temperature for the well block, in °C.

Valid operational range yet to be determined.

Parameters

- **temperature** – The target temperature, in °C.
- **hold_time_minutes** – The number of minutes to hold, after reaching `temperature`, before proceeding to the next command.
- **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.
- **block_max_volume** – The maximum volume of any individual well of the loaded labware. If not supplied, the thermocycler will default to 25µL/well.

New in version 2.0.

set_lid_temperature(*self*, *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.

New in version 2.0.

Useful Types and Definitions

class opentrons.types.**Location**

A location to target as a motion.

The location contains a *Point* (page 88) (in *Deck Coordinates* (page 93)) and possibly an associated *Labware* (page 73) or *Well* (page 76) instance.

It should rarely be constructed directly by the user; rather, it is the return type of most *Well* (page 76) accessors like *Well.top()* (page 77) and is passed directly into a method like `InstrumentContext.aspirate()`.

Warning: The *labware* (page 88) 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 88) attribute.

Warning: The `==` operation compares both the position and associated labware. If you only need to compare locations, compare the *point* (page 88) of each item.

property labware
Alias for field number 1

move(*self*, *point: opentrons.types.Point*) → '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
```

property 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*)

property x
Alias for field number 0

property y
Alias for field number 1

property z
Alias for field number 2

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

Executing and Simulating Protocols

opentrons.execute: functions and entrypoint for running protocols

This module has functions that can be imported to provide protocol contexts for running protocols during interactive sessions like Jupyter or just regular python shells. It also provides a console entrypoint for running a protocol from the command line.

opentrons.execute.**execute**(*protocol_file: <class 'TextIO'>, protocol_name: str, propagate_logs: bool = False, log_level: str = 'warning', emit_runlog: Callable[[Dict[str, Any]], NoneType] = None, custom_labware_paths: List[str] = None, custom_data_paths: List[str] = None*)

Run the protocol itself.

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

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

To call from the command line use either the autogenerated entrypoint `opentrons_execute` or `python -m opentrons.execute`.

If the protocol is using Opentrons Protocol API V1, it does not need to explicitly call `Robot.connect()` or `Robot.discover_modules()`, or `Robot.cache_instrument_models()`.

Parameters

- **protocol_file** (*file-like*) – The protocol file to execute
- **protocol_name** (*str*) – The name of the protocol file. This is required internally, but it may not be a thing we can get from the protocol_file argument.
- **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 emit on the command line.. Default: 'warning'
- **emit_runlog** – A callback for printing the runlog. If specified, this will be called whenever a command adds an entry to the runlog, which can be used for display and progress estimation. If specified, the callback should take a single argument (the name doesn't matter) which will be a dictionary (see below). Default: `None`
- **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`.

The format of the runlog entries is as follows:

```
{
    'name': command_name,
    'payload': {
          'text': string_command_text,
           # The rest of this struct is command-dependent; see
           # opentrons.commands.commands. Its keys match format
           # keys in 'text', so that
           # entry['payload']['text'].format(**entry['payload'])
           # will produce a string with information filled in
```

(continues on next page)

(continued from previous page)

```
    }
}
```

opentrons.execute.**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.

Returns argparse.ArgumentParser The parser with arguments added.

opentrons.execute.**get_protocol_api**(*version: Union[str, opentrons.protocols.types.APIVersion], bundled_labware: Dict[str, ForwardRef('LabwareDefinition')] = None, bundled_data: Dict[str, bytes] = None, extra_labware: Dict[str, ForwardRef('LabwareDefinition')] = None*) → opentrons.protocol_api.protocol_context.ProtocolContext

Build and return a `ProtocolContext` connected to the robot.

This can be used to run protocols from interactive Python sessions such as Jupyter or an interpreter on the command line:

```
>>> from opentrons.execute import get_protocol_api
>>> protocol = get_protocol_api('2.0')
>>> instr = protocol.load_instrument('p300_single', 'right')
>>> instr.home()
```

If `extra_labware` is not specified, any labware definitions saved in the `labware` directory of the Jupyter notebook directory will be available.

When this function is called, modules and instruments will be recached.

Parameters

- **version** – The API version to use. This must be lower than `opentrons.protocol_api.MAX_SUPPORTED_VERSION`. It may be specified either as a string (`'2.0'`) or as a `protocols.types.APIVersion` (`APIVersion(2, 0)`).
- **bundled_labware** – If specified, a mapping from labware names to labware definitions for labware to consider in the protocol. Note that if you specify this, _only_ labware in this argument will be allowed in the protocol. This is preparation for a beta feature and is best not used.
- **bundled_data** – If specified, a mapping from filenames to contents for data to be available in the protocol from `ProtocolContext.bundled_data`.
- **extra_labware** – If specified, a mapping from labware names to labware definitions for labware to consider in the protocol in addition to those stored on the robot. If this is an empty dict, and this function is called on a robot, it will look in the 'labware' subdirectory of the Jupyter data directory for custom labware.

Returns opentrons.protocol_api.ProtocolContext The protocol context.

opentrons.execute.**main**() → int

Handler for command line invocation to run a protocol.

Parameters **argv** – The arguments the program was invoked with; this is usually `sys.argv` but if you want to override that you can.

Returns int A success or failure value suitable for use as a shell return code passed to `sys.exit()` (0 means success, anything else is a kind of failure).

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.

class opentrons.simulate.**AccumulatingHandler**(*level*, *command_queue*)

emit(*self*, *record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class opentrons.simulate.**CommandScraper**(*logger: logging.Logger*, *level: str*, *broker: opentrons.broker.Broker*)

An object that handles scraping the broker for commands

This should be instantiated with the logger to integrate messages from (e.g. `logging.getLogger('opentrons')`), the level to scrape, and the opentrons broker object to subscribe to.

The *commands* (page 91) property contains the list of commands and log messages integrated together. Each element of the list is a dict following the pattern in the docs of *simulate()* (page 92).

property commands

The list of commands. See *simulate()* (page 92)

opentrons.simulate.**allow_bundle**() → bool

Check if bundling is allowed with a special not-exposed-to-the-app flag.

Returns `True` if the environment variable `OT_API_FF_allowBundleCreation` is `"1"`

opentrons.simulate.**bundle_from_sim**(*protocol: opentrons.protocols.types.PythonProtocol*, *context: opentrons.protocol_api.protocol_context.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: List[Mapping[str, 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 92)

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.**get_protocol_api**(*version: Union[str, opentrons.protocols.types.APIVersion]*, *bundled_labware: Dict[str, ForwardRef('LabwareDefinition')]* = *None*, *bundled_data: Dict[str, bytes]* = *None*, *extra_labware: Dict[str, ForwardRef('LabwareDefinition')]* = *None*, *hardware_simulator: Union[opentrons.hardware_control.thread_manager.ThreadManager, opentrons.hardware_control.adapters.SynchronousAdapter, ForwardRef('HasLoop')]* = *None*) → opentrons.protocol_api.protocol_context.ProtocolContext

Build and return a `ProtocolContext` connected to Virtual Smoothie.

This can be used to run protocols from interactive Python sessions such as Jupyter or an interpreter on the command line:

```
>>> from opentrons.simulate import get_protocol_api
>>> protocol = get_protocol_api('2.0')
>>> instr = protocol.load_instrument('p300_single', 'right')
>>> instr.home()
```

If `extra_labware` is not specified, any labware definitions saved in the `labware` directory of the Jupyter notebook directory will be available.

Parameters

- **`version`** – The API version to use. This must be lower than `opentrons.protocol_api.MAX_SUPPORTED_VERSION`. It may be specified either as a string (`'2.0'`) or as a `protocols.types.APIVersion` (`APIVersion(2, 0)`).
- **`bundled_labware`** – If specified, a mapping from labware names to labware definitions for labware to consider in the protocol. Note that if you specify this, _only_ labware in this argument will be allowed in the protocol. This is preparation for a beta feature and is best not used.
- **`bundled_data`** – If specified, a mapping from filenames to contents for data to be available in the protocol from `ProtocolContext.bundled_data`.
- **`extra_labware`** – If specified, a mapping from labware names to labware definitions for labware to consider in the protocol in addition to those stored on the robot. If this is an empty dict, and this function is called on a robot, it will look in the 'labware' subdirectory of the Jupyter data directory for custom labware.
- **`hardware_simulator`** – If specified, a hardware simulator instance.

Returns opentrons.protocol_api.ProtocolContext The protocol context.

`opentrons.simulate.`**`main`**`()` → int

Run the simulation

`opentrons.simulate.`**`simulate`**(*protocol_file: <class 'TextIO'>*, *file_name: str = None*, *custom_labware_paths: List[str] = None*, *custom_data_paths: List[str] = None*, *propagate_logs: bool = False*, *hardware_simulator_file_path: str = None*, *log_level: str = 'warning'*) → Tuple[List[Mapping[str, Any]], 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`.
- **hardware_simulator_file_path** – A path to a JSON file defining a hardware simulator.
- **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 bundling is allowed (see `allow_bundling()`) and this is an unbundled Protocol API v2 python protocol. In other cases it is None.

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 68), in addition to being an instance of *opentrons.protocol_api.labware.Well* (page 76) 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 88), which can be created in the same way.

5.9 Examples

All examples on this page use a `'corning_96_wellplate_360ul_flat'` (an ANSI standard 96-well plate[48]) in slot 1, and two `'opentrons_96_tiprack_300ul'` (the Opentrons standard 300 µL tiprack[49]) 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 reservoir[50]) in slot 4.

---

[48] https://labware.opentrons.com/corning_96_wellplate_360ul_flat
[49] https://labware.opentrons.com/opentrons_96_tiprack_300ul
[50] https://labware.opentrons.com/usascientific_12_reservoir_22ml

Basic Transfer

Moving 100 µL from one well to another:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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

metadata = {'apiLevel': '2.7'}

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('B1'))
    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 `7`, and uses that loop's current value to transfer from all wells in a reservoir to each row of a plate:

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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)
    reservoir = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])
    # distribute 20uL from reservoir:A1 -> plate:row:1
    # distribute 20uL from reservoir: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, reservoir.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 the first five wells in the reservoir, while creating an air gap between each sample.

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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)
    reservoir = 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 reservoir.wells()[:4]:
        p300.aspirate(35, well)
        p300.air_gap(10)

    p300.dispense(225, plate['A1'])

    p300.return_tip()
```

Dilution

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

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.7'}

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)
    reservoir = protocol.load_labware('usascientific_12_reservoir_22ml', 4)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1,␣
↪tiprack_2])
    p300.distribute(50, reservoir['A12'], plate.wells())  # dilutent

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

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

        # transfer 30uL of source to first well in column
        p300.transfer(30, source, row[0], mix_after=(3, 25))

        # dilute the sample down the column
```

(continues on next page)

(continued from previous page)

```
        p300.transfer(
            30, row[:11], row[1:],
            mix_after=(3, 25))
```

Plate Mapping

This example deposits 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

metadata = {'apiLevel': '2.7'}

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)
    reservoir = 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, reservoir['A12'], plate.wells())
```

5.10 Advanced Control

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 OT-2 without using the Opentrons App.

Jupyter Notebook

The OT-2 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 OT-2's Jupyter Notebook by following these steps:

1. Open your Opentrons App and look for the IP address of your OT-2 on the information page.
2. Type in `(Your OT-2'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 OT-2 itself. These protocols will only be on the OT-2 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 your protocol - 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()`* (page 90), into which you pass the same API version (see *Versioning* (page 9)) that you would specify in your protocol's metadata:

```
import opentrons.execute
protocol = opentrons.execute.get_protocol_api('2.7')
protocol.home()
```

This returns the same kind of object - a *`ProtocolContext`* (page 58) - 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 58) object is available in the other sections of this guide - *Pipettes* (page 27), *Building Block Commands* (page 35), *Complex Commands* (page 45), *Labware* (page 12), and *Hardware Modules* (page 17); a full list of all available attributes and methods is available in *API Version 2 Reference* (page 58).

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 58) 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 58), 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.

Note: Before you can command the OT-2 to move using the protocol API you have just built, you must home the robot using `protocol.home()`. If you try to move the OT-2 before you have called `protocol.home()`, you will get a `MustHomeError`.

Running A Previously-Written Protocol

If you have a protocol that you have already written you can run it directly in Jupyter. Copy the protocol into a cell and execute it - this won't cause the OT-2 to move, it just makes the function available. Then, call the `run` function you just defined, and give it a *`ProtocolContext`* (page 58):

Custom Labware

If you have custom labware definitions you want to use with Jupyter, make a new directory called "labware" in Jupyter and put the definitions there. These definitions will be available when you call `load_labware`.

Command Line

The OT-2's command line is accessible either by creating a new terminal in Jupyter or by using SSH to access its terminal[51].

To execute a protocol via SSH, copy it to the OT-2 using a program like `scp` and then 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.

[51] https://support.opentrons.com/en/articles/3203681

Python Module Index

O

Index

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z