Welcome

The Opentrons Python Protocol API is a Python framework designed to make it easy to write automated biology lab protocols. Python protocols can control Opentrons Flex and OT-2 robots, their pipettes, and optional hardware modules. We've designed the API to be accessible to anyone with basic Python and wet-lab skills.

As a bench scientist, you should be able to code your protocols in a way that reads like a lab notebook. You can write a fully functional protocol just by listing the equipment you'll use (modules, labware, and pipettes) and the exact sequence of movements the robot should make.

As a programmer, you can leverage the full power of Python for advanced automation in your protocols. Perform calculations, manage external data, use built-in and imported Python modules, and more to implement your custom lab workflow.

Getting Started

New to Python protocols? Check out the Tutorial to learn about the different parts of a protocol file and build a working protocol from scratch.

If you want to **dive right into code**, take a look at our Protocol Examples and the comprehensive API Version 2 Reference.

When you're ready to **try out a protocol**, download the Opentrons App, import the protocol file, and run it on your robot.

How the API Works

The design goal of this API is to make code readable and easy to understand. A protocol, in its most basic form:

1. Provides some information about who made the protocol and what it is for.
2. Specifies which type of robot the protocol should run on.
3. Tells the robot where to find labware, pipettes, and (optionally) hardware modules.
4. Commands the robot to manipulate its attached hardware.

For example, if we wanted to transfer liquid from well A1 to well B1 on a plate, our protocol would look like:

Flex | **OT-2**

```
from opentrons import protocol_api

# metadata
metadata = {
    "protocolName": "My Protocol",
    "author": "Name <opentrons@example.com>",
    "description": "Simple protocol to get started using the OT-2",
}

# requirements
requirements = {"robotType": "OT-2", "apiLevel": "2.14"}

# protocol run function
def run(protocol: protocol_api.ProtocolContext):
    # labware
    plate = protocol.load_labware(
```

👋 Hi there! What brings you to Opentrons today?
1

```
    # pipettes
    left_pipette = protocol.load_instrument(
        "p300_single", mount="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()
```

This example proceeds completely linearly. Following it line-by-line, you can see that it has the following effects:

1. Gives the name, contact information, and a brief description for the protocol.
2. Indicates the protocol should run on an OT-2 robot, using API version 2.14.
3. Tells the robot that there is:
    a. A 96-well flat plate in slot 1.
    b. A rack of 300 µL tips in slot 2.
    c. A single-channel 300 µL pipette attached to the left mount, which should pick up tips from the afore-mentioned rack.
4. Tells the robot to act by:
    a. Picking up the first tip from the tip rack.
    b. Aspirating 100 µL of liquid from well A1 of the plate.
    c. Dispensing 100 µL of liquid into well B1 of the plate.
    d. Dropping the tip in the trash.

There is much more that Opentrons robots and the API can do! The Building Block Commands, Complex Commands, and Hardware Modules pages cover many of these functions.

More Resources

Opentrons App

The Opentrons App is the easiest way to run your Python protocols. The app supports the latest versions of macOS, Windows, and Ubuntu.

Support

Questions about setting up your robot, using Opentrons software, or troubleshooting? Check out our support articles or get in touch directly with Opentrons Support.

Custom Protocol Service

Don't have the time or resources to write your own protocols? The Opentrons Custom Protocols service can get you set up in as little as a week.

Contributing

Opentrons software, including the Python API and this documentation, is open source. If you have an improvement or an interesting idea, you can create an issue on GitHub by following our guidelines.

That guide also includes more information on how to directly contribute code.

opentrons

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Tutorial

Introduction

This tutorial will guide you through creating a Python protocol file from scratch. At the end of this process you'll have a complete protocol that can run on a Flex or an OT-2 robot. If you don't have a Flex or an OT-2 (or if you're away from your lab, or if your robot is in use), you can use the same file to simulate the protocol on your computer instead.

What You'll Automate

The lab task that you'll automate in this tutorial is *serial dilution*: taking a solution and progressively diluting it by transferring it stepwise across a plate from column 1 to column 12. With just a dozen or so lines of code, you can instruct your robot to perform the hundreds of individual pipetting actions necessary to fill an entire 96-well plate. And all of those liquid transfers will be done automatically, so you'll have more time to do other work in your lab.

Before You Begin

You're going to write some Python code, but you don't need to be a Python expert to get started writing Opentrons protocols. You should know some basic Python syntax, like how it uses indentation to group blocks of code, dot notation for calling methods, and the format of lists and dictionaries. You'll also be using common control structures like `if` statements and `for` loops.

To run your code, make sure that you've installed Python 3 and the pip package installer. You should write your code in your favorite plaintext editor or development environment and save it in a file with a `.py` extension, like `dilution-tutorial.py`.

Hardware and Labware

Before running a protocol, you'll want to have the right kind of hardware and labware ready for your Flex or OT-2.

- **Flex users** should review Chapter 2: Installation and Relocation in the instruction manual. Specifically, see the pipette information in the "Instrument Installation and Calibration" section. You can use either a 1-channel or 8-channel pipette for this tutorial. Most Flex code examples will use a Flex 1-Channel 1000 µL pipette.
- **OT-2 users** should review the robot setup and pipette information on the Get Started page. Specifically, see attaching pipettes and initial calibration. You can use either a single-channel or 8-channel pipette for this tutorial. Most OT-2 code examples will use a P300 Single-Channel GEN2 pipette.

The Flex and OT-2 use similar labware for serial dilution. The tutorial code will use the labware listed in the table below, but as long as you have labware of each type you can modify the code to run with your labware.

Labware type	Labware name	API load name
Reservoir	NEST 12 Well Reservoir 15 mL	`nest_12_reservoir`
Well plate	NEST 96 Well Plate 200 µL Flat	`nest_96_wellplate_`
Flex tip rack	Opentrons Flex Tips, 200 µL	`opentrons_flex_96_tiprack_200ul`

👋 Hi there! What brings you to Opentrons today?
1

For the liquids, you can use plain water as the diluent and water dyed with food coloring as the solution.

Create a Protocol File

Let's start from scratch to create your serial dilution protocol. Open up a new file in your editor and start with the line:

```
from opentrons import protocol_api
```

Throughout this documentation, you'll see protocols that begin with the `import` statement shown above. It identifies your code as an Opentrons protocol. This statement is not required, but including it is a good practice and allows most code editors to provide helpful autocomplete suggestions.

Everything else in the protocol file is required. Next, you'll specify the version of the API you're using. Then comes the core of the protocol: defining a single `run()` function that provides the locations of your labware, states which kind of pipettes you'll use, and finally issues the commands that the robot will perform.

For this tutorial, you'll write very little Python outside of the `run()` function. But for more complex applications it's worth remembering that your protocol file *is* a Python script, so any Python code that can run on your robot can be a part of a protocol.

Metadata

Every protocol needs to have a metadata dictionary with information about the protocol. At minimum, you need to specify what version of the API the protocol requires. The scripts for this tutorial were validated against API version 2.15, so specify:

```
metadata = {'apiLevel': '2.15'}
```

You can include any other information you like in the metadata dictionary. The fields `protocolName`, `description`, and `author` are all displayed in the Opentrons App, so it's a good idea to expand the dictionary to include them:

```
metadata = {
    'apiLevel': '2.15',
    'protocolName': 'Serial Dilution Tutorial',
    'description': '''This protocol is the outcome of following the
                   Python Protocol API Tutorial located at
                   https://docs.opentrons.com/v2/tutorial.html. It takes a
                   solution and progressively dilutes it by transferring it
                   stepwise across a plate.''',
    'author': 'New API User'
    }
```

Note, if you have a Flex, or are using an OT-2 with API v2.15 (or higher), we recommend adding a `requirements` section to your code. See the Requirements section below.

Requirements

The `requirements` code block can appear before *or* after the `metadata` code block in a Python protocol. It uses the following syntax and accepts two arguments: `robotType` and `apiLevel`.

Whether you need a `requirements` block depends on your robot model and API version.

- **Flex:** The `requirements` block is always required. And, the API version does not go in the `metadata` section. The API version belongs in the `requirements`. For example:

  ```
  requirements = {"robotType": "Flex", "apiLevel": "2.15"}
  ```

```
    requirements = {"robotType": "OT-2", "apiLevel": "2.15"}
```

With the metadata and requirements defined, you can move on to creating the `run()` function for your protocol.

The `run()` function

Now it's time to actually instruct the Flex or OT-2 how to perform serial dilution. All of this information is contained in a single Python function, which has to be named `run`. This function takes one argument, which is the *protocol context*. Many examples in these docs use the argument name `protocol`, and sometimes they specify the argument's type:

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

With the protocol context argument named and typed, you can start calling methods on `protocol` to add labware and hardware.

Labware

For serial dilution, you need to load a tip rack, reservoir, and 96-well plate on the deck of your Flex or OT-2. Loading labware is done with the **`load_labware()`** method of the protocol context, which takes two arguments: the standard labware name as defined in the Opentrons Labware Library, and the position where you'll place the labware on the robot's deck.

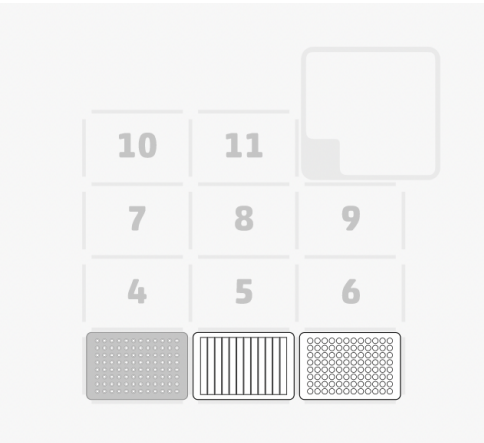
Flex | **OT-2**

Here's how to load the labware on an OT-2 in slots 1, 2, and 3 (repeating the `def` statement from above to show proper indenting):

```
def run(protocol: protocol_api.ProtocolContext):
    tips = protocol.load_labware('opentrons_96_tiprack_300ul', 1)
    reservoir = protocol.load_labware('nest_12_reservoir_15ml', 2)
    plate = protocol.load_labware('nest_96_wellplate_200ul_flat', 3)
```

If you're using a different model of labware, find its name in the Labware Library and replace it in your code.

Now the robot will expect to find labware in a configuration that looks like this:



You may notice that these deck maps don't show where the liquids will be at the start of the protocol. Liquid definitions aren't required in Python protocols, unlike protocols made in Protocol Designer. If you want to identify liquids, see Labeling Liquids in Wells. (Sneak peek: you'll put the diluent in column 1 of the reservoir and the solution in column 2 of the reservoir.)

takes three arguments: the name of the pipette, the mount it's installed in, and the tip racks it should use when performing transfers. Load whatever pipette you have installed in your robot by using its standard pipette name. Here's how to load the pipette in the left mount and instantiate it as a variable named `left_pipette`:

```
# Flex
left_pipette = protocol.load_instrument('flex_1channel_1000', 'left', tip_racks=[tips])

# OT-2
left_pipette = protocol.load_instrument('p300_single_gen2', 'left', tip_racks=[tips])
```

Since the pipette is so fundamental to the protocol, it might seem like you should have specified it first. But there's a good reason why pipettes are loaded after labware: you need to have already loaded `tips` in order to tell the pipette to use it. And now you won't have to reference `tips` again in your code — it's assigned to the `left_pipette` and the robot will know to use it when commanded to pick up tips.

> **Note:**
>
> You may notice that the value of `tip_racks` is in brackets, indicating that it's a list. This serial dilution protocol only uses one tip rack, but some protocols require more tips, so you can assign them to a pipette all at once, like `tip_racks=[tips1, tips2]`.

Commands

Finally, all of your labware and hardware is in place, so it's time to give the robot pipetting commands. The required steps of the serial dilution process break down into three main phases:

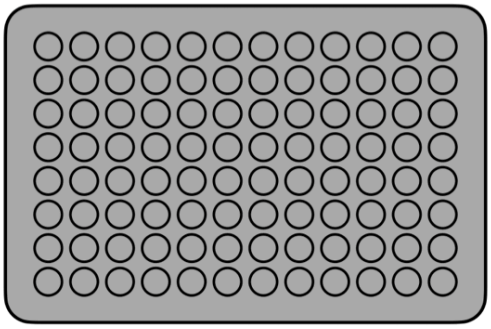
1. Measure out equal amounts of diluent from the reservoir to every well on the plate.
2. Measure out equal amounts of solution from the reservoir into wells in the first column of the plate.
3. Move a portion of the combined liquid from column 1 to 2, then from column 2 to 3, and so on all the way to column 12.

Thanks to the flexibility of the API's **`transfer()`** method, which combines many building block commands into one call, each of these phases can be accomplished with a single line of code! You'll just have to write a few more lines of code to repeat the process for as many rows as you want to fill.

Let's start with the diluent. This phase takes a larger quantity of liquid and spreads it equally to many wells. `transfer()` can handle this all at once, because it accepts either a single well or a list of wells for its source and destination:

```
left_pipette.transfer(100, reservoir['A1'], plate.wells())
```

Breaking down these single lines of code shows the power of complex commands. The first argument is the amount to transfer to each destination, 100 µL. The second argument is the source, column 1 of the reservoir (which is still specified with grid-style coordinates as `A1` — a reservoir only has an A row). The third argument is the destination. Here, calling the **`wells()`** method of `plate` returns a list of *every well*, and the command will apply to all of them.



In plain English, you've instructed the robot, "For every well on the plate, aspirate 100 µL of fluid from column 1 of the reservoir and dispense it in the well." That's how we understand this line of code as scientists, yet the robot will understand and execute it as nearly 200 discrete actions.

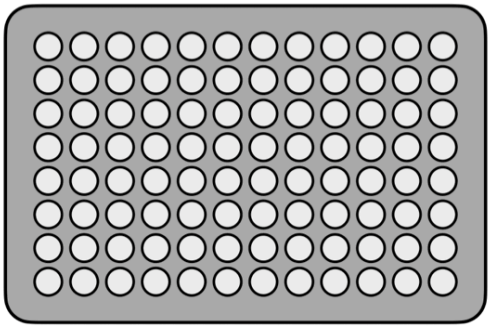
Now it's time to start mixing in the solution. To do this row by row, nest the commands in a `for` loop:

```
for i in range(8):
    row = plate.rows()[i]
```

Using Python's built-in `range` class is an easy way to repeat this block 8 times, once for each row. This also lets you use the repeat index `i` with `plate.rows()` to keep track of the current row.

```
for i in range(8):
    row = plate.rows()[i]
    left_pipette.transfer(100, reservoir['A2'], row[0], mix_after=(3, 50))
    left_pipette.transfer(100, row[:11], row[1:], mix_after=(3, 50))
```

i = 0

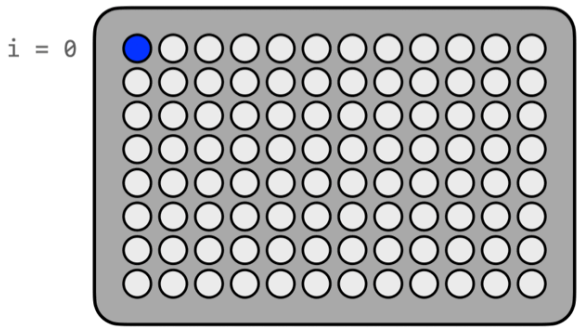


In each row, you first need to add solution. This will be similar to what you did with the diluent, but putting it only in column 1 of the plate. It's best to mix the combined solution and diluent thoroughly, so add the optional `mix_after` argument to `transfer()`:

```
left_pipette.transfer(100, reservoir['A2'], row[0], mix_after(3, 50))
```

so on. The fourth argument specifies to mix 3 times with 50 µL of fluid each time.

```
for i in range(8):
    row = plate.rows()[i]
    left_pipette.transfer(100, reservoir['A2'], row[0], mix_after=(3, 50))
    left_pipette.transfer(100, row[:11], row[1:], mix_after=(3, 50))
```

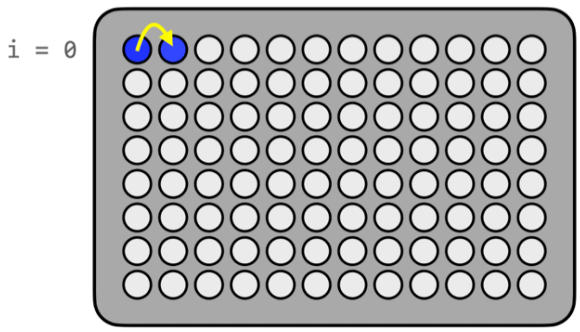


Finally, it's time to dilute the solution down the row. One approach would be to nest another `for` loop here, but instead let's use another feature of the `transfer()` method, taking lists as the source and destination arguments:

```
left_pipette.transfer(100, row[:11], row[1:], mix_after(3, 50))
```

There's some Python shorthand here, so let's unpack it. You can get a range of indices from a list using the colon `:` operator, and omitting it at either end means "from the beginning" or "until the end" of the list. So the source is `row[:11]`, from the beginning of the row until its 11th item. And the destination is `row[1:]`, from index 1 (column 2!) until the end. Since both of these lists have 11 items, `transfer()` will *step through them in parallel*, and they're constructed so when the source is 0, the destination is 1; when the source is 1, the destination is 2; and so on. This condenses all of the subsequent transfers down the row into a single line of code.

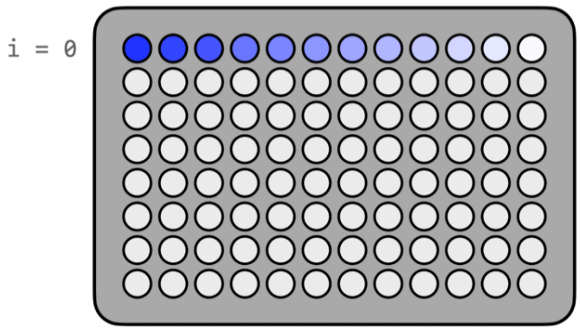
```
for i in range(8):
    row = plate.rows()[i]
    left_pipette.transfer(100, reservoir['A2'], row[0], mix_after=(3, 50))
    left_pipette.transfer(100, row[:11], row[1:], mix_after=(3, 50))
```




```
row[:11] = row[0], row[1], row[2], ... row[10]
row[1:]  = row[1], row[2], row[3], ... row[11]
```

All that remains is for the loop to repeat these steps, filling each row down the plate.

```
left_pipette.transfer(100, row[:11], row[1:], mix_after=(3, 50))
```



That's it! If you're using a single-channel pipette, you're ready to try out your protocol.

8-Channel Pipette

If you're using an 8-channel pipette, you'll need to make a couple tweaks to the single-channel code from above. Most importantly, whenever you target a well in row A of a plate with an 8-channel pipette, it will move its topmost tip to row A, lining itself up over the entire column.

Thus, when adding the diluent, instead of targeting every well on the plate, you should only target the top row:

```
left_pipette.transfer(100, reservoir['A1'], plate.rows()[0])
```

And by accessing an entire column at once, the 8-channel pipette effectively implements the `for` loop in hardware, so you'll need to remove it:

```
row = plate.rows()[0]
left_pipette.transfer(100, reservoir['A2'], row[0], mix_after=(3, 50))
left_pipette.transfer(100, row[:11], row[1:], mix_after=(3, 50))
```

Instead of tracking the current row in the `row` variable, this code sets it to always be row A (index 0).

Try Your Protocol

There are two ways to try out your protocol: simulation on your computer, or a live run on a Flex or OT-2. Even if you plan to run your protocol on a robot, it's a good idea to check the simulation output first.

If you get any errors in simulation, or you don't get the outcome you expected when running your protocol, you can check your code against these reference protocols on GitHub:

- Flex: Single-channel serial dilution
- Flex: 8-channel serial dilution
- OT-2: Single-channel serial dilution
- OT-2: 8-channel serial dilution

In Simulation

Simulation doesn't require having a robot connected to your computer. You just need to install the Opentrons Python module from Pip (`pip install opentrons`). This will give you access to the `opentrons_simulate` command-line utility

cation of your saved protocol file and run:

```
$ opentrons_simulate dilution-tutorial.py
```

This should generate a lot of output! As written, the protocol has about 1000 steps. If you're curious how long that will take, you can use an experimental feature to estimate the time:

```
$ opentrons_simulate dilution-tutorial.py -e -o nothing
```

The `-e` flag estimates duration, and `-o nothing` suppresses printing the run log. This indicates that using a single-channel pipette for serial dilution across the whole plate will take about half an hour — plenty of time to grab a coffee while your robot pipettes for you!

If that's too long, you can always cancel your run partway through or modify `for i in range(8)` to loop through fewer rows.

On a Robot

The simplest way to run your protocol on a Flex or OT-2 is to use the Opentrons App. When you first launch the Opentrons App, you will see the Protocols screen. (Click **Protocols** in the left sidebar to access it at any other time.) Click **Import** in the top right corner to reveal the Import a Protocol pane. Then click **Choose File** and find your protocol in the system file picker, or drag and drop your protocol file into the well.

You should see "Protocol - Serial Dilution Tutorial" (or whatever `protocolName` you entered in the metadata) in the list of protocols. Click the three-dot menu (⋮) for your protocol and choose **Start setup**.

If you have any remaining calibration tasks to do, you can finish them up here. Below the calibration section is a preview of the initial deck state. Optionally you can run Labware Position Check, or you can go ahead and click **Proceed to Run**.

On the Run tab, you can double-check the Run Preview, which is similar to the command-line simulation output. Make sure all your labware and liquids are in the right place, and then click **Start run**. The run log will update in real time as your robot proceeds through the steps.

When it's all done, check the results of your serial dilution procedure — you should have a beautiful dye gradient running across the plate!

Labware

Labware are the durable or consumable items that you work with, reuse, or discard while running a protocol on a Flex or OT-2. Items such as pipette tips, well plates, tubes, and reservoirs are all examples of labware. This section provides a brief overview of default labware, custom labware, and how to use basic labware API methods when creating a protocol for your robot.

> **Note:**
>
> Code snippets use coordinate deck slot locations (e.g. `'D1'`, `'D2'`), like those found on Flex. If you have an OT-2 and are using API version 2.14 or earlier, replace the coordinate with its numeric OT-2 equivalent. For example, slot D1 on Flex corresponds to slot 1 on an OT-2. See Deck Slots for more information.

Labware Types

Default Labware

Default labware is everything listed in the Opentrons Labware Library. When used in a protocol, your Flex or OT-2 knows how to work with default labware. However, you must first inform the API about the labware you will place on the robot's deck. Search the library when you're looking for the API load names of the labware you want to use. You can copy the load names from the library and pass them to the `load_labware()` method in your protocol.

Custom Labware

Custom labware is labware that is not listed the Labware Library. If your protocol needs something that's not in the library, you can create it with the Opentrons Labware Creator. However, before using the Labware Creator, you should take a moment to review the support article Creating Custom Labware Definitions.

After you've created your labware, save it as a `.json` file and add it to the Opentrons App. See Using Labware in Your Protocols for instructions.

If other people need to use your custom labware definition, they must also add it to their Opentrons App.

Loading Labware

Throughout this section, we'll use the labware listed in the following table.

Labware type	Labware name	API load name
Well plate	Corning 96 Well Plate 360 µL Flat	`corning_96_wellplate_360ul_flat`
Flex tip rack	Opentrons Flex 96 Tips 200 µL	`opentrons_flex_96_tiprack_200ul`
OT-2 tip rack	Opentrons 96 Tip Rack 300 µL	`opentrons_96_tiprack_300ul`

```
#Flex
tiprack = protocol.load_labware('opentrons_flex_96_tiprack_200ul', 'D1')
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 'D2')


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

New in version 2.0.

When the `load_labware` method loads labware into your protocol, it returns a **`Labware`** object.

> **Tip:**
>
> The `load_labware` method includes an optional `label` argument. You can use it to identify labware with a descriptive name. If used, the label value is displayed in the Opentrons App. For example:
>
> ```
> tiprack = protocol.load_labware(
>     load_name='corning_96_wellplate_360ul_flat',
>     location='D1',
>     label='any-name-you-want')
> ```

Loading Labware on Adapters

The previous section demonstrates loading labware directly into a deck slot. But you can also load labware on top of an adapter that either fits on a module or goes directly on the deck. The ability to combine labware with adapters adds functionality and flexibility to your robot and protocols.

You can either load the adapter first and the labware second, or load both the adapter and labware all at once.

Loading Separately

The `load_adapter()` method is available on `ProtocolContext` and module contexts. It behaves similarly to `load_labware()`, requiring the load name and location for the desired adapter. Load a module, adapter, and labware with separate calls to specify each layer of the physical stack of components individually:

```
hs_mod = protocol.load_module('heaterShakerModuleV1', 'D1')
hs_adapter = hs_mod.load_adapter('opentrons_96_flat_bottom_adapter')
hs_plate = hs_adapter.load_labware('nest_96_wellplate_200ul_flat')
```

New in version 2.15: The `load_adapter()` method.

Loading Together

Use the `adapter` argument of `load_labware()` to load an adapter at the same time as labware. For example, to load the same 96-well plate and adapter from the previous section at once:

```
hs_plate = hs_mod.load_labware(
    name='nest_96_wellplate_200ul_flat',
    adapter='opentrons_96_flat_bottom_adapter'
)
```

New in version 2.15: The `adapter` parameter.

The API also has some “combination” labware definitions, which treat the adapter and labware as a unit:

```
hs_combo = hs_mod.load_labware(
    'opentrons_96_flat_bottom_adapter_nest_wellplate_200ul_flat'
)
```

Accessing Wells in Labware

Well Ordering

You need to select which wells to transfer liquids to and from over the course of a protocol.

Rows of wells on a labware have labels that are capital letters starting with A. For instance, an 96-well plate has 8 rows, labeled `'A'` through `'H'`.

Columns of wells on a labware have labels that are numbers starting with 1. For instance, a 96-well plate has columns `'1'` through `'12'`.

All well-accessing functions start with the well at the top left corner of the labware. The ending well is in the bottom right. The order of travel from top left to bottom right depends on which function you use.

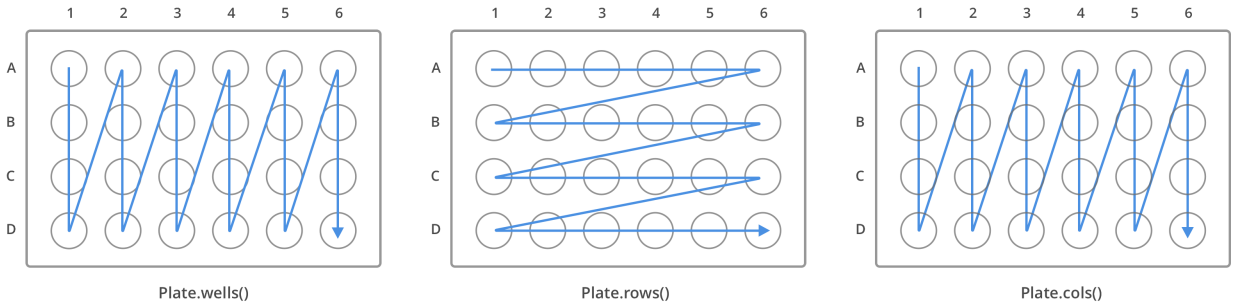


Plate.wells() Plate.rows() Plate.cols()

The code in this section assumes that `plate` is a 24-well plate. For example:

```
plate = protocol.load_labware('corning_24_wellplate_3.4ml_flat', location='D1')
```

Accessor Methods

The API provides 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	Returns	Example
`Labware.wells()`	List of all wells.	`[labware:A1, labware:B1, labware:C1...]`
`Labware.rows()`	List of lists grouped by row.	`[[labware:A1, labware:A2...], [labware:B1, labware:B2...]]`
`Labware.columns()`	List of lists grouped by column.	`[[labware:A1, labware:B1...], [labware:A2, labware:B2...]]`
`Labware.wells_by_name()`	Dictionary with well names as keys.	`{'A1': labware:A1, 'B1': labware:B1}`
`Labware.rows_by_name()`	Dictionary with row names as keys.	`{'A': [labware:A1, labware:A2...], 'B': [labware:B1, labware:B2...]}`
`Labware.columns_by_name()`	Dictionary with column names as keys.	`{'1': [labware:A1, labware:B1...], '2': [labware:A2, labware:B2...]}`

Accessing Individual Wells

Dictionary Access

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

If a well does not exist in the labware, such as `plate['H12']` on a 24-well plate, the API will raise a `KeyError`. In contrast, it would be a valid reference on a standard 96-well plate.

New in version 2.0.

List Access From `wells`

In addition to referencing wells by name, you can also reference them with zero-indexing. The first well in a labware is at position 0.

```
plate.wells()[0]   # well A1
plate.wells()[23]  # well D6
```

> **Tip:**
>
> You may find coordinate well names like `"B3"` easier to reason with, especially when working with irregular labware, e.g. `opentrons_10_tuberack_falcon_4x50ml_6x15ml_conical` (see the Opentrons 10 Tube Rack in the Labware Library). 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 handling liquid, you can provide a group of wells as the source or destination. Alternatively, you can take a group of wells and loop (or iterate) through them, with each liquid-handling command inside the loop accessing the loop index.

Use **`Labware.rows_by_name()`** to access a specific row of wells or **`Labware.columns_by_name()`** to access a specific column of wells 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')  # Column "1" has 4 wells
print('Row "A" has', len(row_dict), '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 transfer 50 µL of liquid from the first well of a reservoir to each of the wells of row `'A'` on a plate:

```
for well in plate.rows()[0]:
    pipette.transfer(reservoir['A1'], well, 50)
```

Equivalently, using `rows_by_name`:

```
for well in plate.rows_by_name()['A'].values():
    pipette.transfer(reservoir['A1'], well, 50)
```

New in version 2.0.

Labeling Liquids in Wells

- For Flex protocols, on the touchscreen.
- For Flex or OT-2 protocols, in the Opentrons App (v6.3.0 or higher).

To use these optional methods, first create a liquid object with **`ProtocolContext.define_liquid()`** and then label individual wells by calling **`Well.load_liquid()`**.

Let's examine how these two methods work. The following examples demonstrate how to define colored water samples for a well plate and reservoir.

Defining Liquids

This example uses `define_liquid` to create two liquid objects and instantiates them with the variables `greenWater` and `blueWater`, respectively. The arguments for `define_liquid` are all required, and let you name the liquid, describe it, and assign it a color:

```
greenWater = protocol.define_liquid(
    name="Green water",
    description="Green colored water for demo",
    display_color="#00FF00",
)
blueWater = protocol.define_liquid(
    name="Blue water",
    description="Blue colored water for demo",
    display_color="#0000FF",
)
```

New in version 2.14.

The `display_color` parameter accepts a hex color code, which adds a color to that liquid's label when you import your protocol into the Opentrons App. The `define_liquid` method accepts standard 3-, 4-, 6-, and 8-character hex color codes.

Labeling Wells and Reservoirs

This example uses `load_liquid` to label the initial well location, contents, and volume (in µL) for the liquid objects created by `define_liquid`. Notice how values of the `liquid` argument use the variable names `greenWater` and `blueWater` (defined above) to associate each well with a particular liquid:

```
well_plate["A1"].load_liquid(liquid=greenWater, volume=50)
well_plate["A2"].load_liquid(liquid=greenWater, volume=50)
well_plate["B1"].load_liquid(liquid=blueWater, volume=50)
well_plate["B2"].load_liquid(liquid=blueWater, volume=50)
reservoir["A1"].load_liquid(liquid=greenWater, volume=200)
reservoir["A2"].load_liquid(liquid=blueWater, volume=200)
```

New in version 2.14.

This information is available after you import your protocol to the app or send it to Flex. A summary of liquids appears on the protocol detail page, and well-by-well detail is available on the run setup page (under Initial Liquid Setup in the app, or under Liquids on Flex).

> **Note:**
>
> `load_liquid` does not validate volume for your labware nor does it prevent you from adding multiple liquids to each well. For example, you could label a 40 µL well with `greenWater`, `volume=50`, and then also add blue water to the well. The API won't stop you. It's your responsibility to ensure the labels you use accurately reflect the amounts and types of liquid you plan to place into wells and reservoirs.

function that manipulates liquids. It only tells you how much liquid should be in a well at the start of the protocol. You need to use a method like `transfer()` to physically move liquids from a source to a destination.

Well Dimensions

The functions in the Accessing Wells in Labware section above return a single `Well` object or a larger object representing many wells. `Well` objects have attributes that provide information about their physical shape, such as the depth or diameter, as specified in their corresponding labware definition. These properties can be used for different applications, such as calculating the volume of a well or a position relative to the well.

Depth

Use `Well.depth` to get the distance in mm between the very top of the well and the very bottom. For example, a conical well's depth is measured from the top center to the bottom center of the well.

```
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 'D1')
depth = plate['A1'].depth  # 10.67
```

Diameter

Use `Well.diameter` to get the diameter of a given well in mm. Since diameter is a circular measurement, this attribute is only present on labware with circular wells. If the well is not circular, the value will be `None`. Use length and width (see below) for non-circular wells.

```
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 'D1')
diameter = plate['A1'].diameter      # 6.86
```

Length

Use `Well.length` to get the length of a given well in mm. Length is defined as the distance along the robot's x-axis (left to right). This attribute is only present on rectangular wells. If the well is not rectangular, the value will be `None`. Use diameter (see above) for circular wells.

```
plate = protocol.load_labware('nest_12_reservoir_15ml', 'D1')
length = plate['A1'].length  # 8.2
```

Width

Use `Well.width` to get the width of a given well in mm. Width is defined as the distance along the y-axis (front to back). This attribute is only present on rectangular wells. If the well is not rectangular, the value will be `None`. Use diameter (see above) for circular wells.

```
plate = protocol.load_labware('nest_12_reservoir_15ml', 'D1')
width = plate['A1'].width  # 71.2
```

New in version 2.9.

Sign Up For Our Newsletter

Email**

Heater-Shaker Module

The Heater-Shaker Module provides on-deck heating and orbital shaking. The module can heat from 37 to 95 °C, and can shake samples from 200 to 3000 rpm.

The Heater-Shaker Module is represented in code by a `HeaterShakerContext` object. For example:

```
hs_mod = protocol.load_module(
    module_name="heaterShakerModuleV1", location="D1"
)
```

New in version 2.13.

Deck Slots

The supported deck slot positions for the Heater-Shaker depend on the robot you're using.

Robot Model	Heater-Shaker Deck Placement
Flex	In any deck slot in column 1 or 3. The module can go in slot A3, but you need to move the trash bin first.
OT-2	In deck slot 1, 3, 4, 6, 7, or 10.

OT-2 Placement Restrictions

On OT-2, you need to restrict placement of other modules and labware around the Heater-Shaker. On Flex, the module is installed below-deck in a caddy and there is more space between deck slots, so these restrictions don't apply.

In general, it's best to leave all slots adjacent to the Heater-Shaker empty. If your protocol requires filling those slots, observe the following restrictions to avoid physical crashes involving the Heater-Shaker.

Adjacent Modules

Do not place other modules next to the Heater-Shaker. Keeping adjacent deck slots clear helps prevents collisions during shaking and while opening the labware latch. Loading a module next to the Heater-Shaker on OT-2 will raise a `DeckConflictError`.

Tall Labware

Do not place labware taller than 53 mm to the left or right of the Heater-Shaker. This prevents the Heater-Shaker's latch from colliding with the adjacent labware. Common labware that exceed the height limit include Opentrons tube racks and Opentrons 1000 µL tip racks. Loading tall labware to the right or left of the Heater-Shaker on OT-2 will raise a `DeckConflictError`.

8-Channel Pipettes

There is one exception: to the front or back of the Heater-Shaker, an 8-channel pipette can access tip racks only. Attempting to pipette to non-tip-rack labware will also raise a `PipetteMovementRestrictedByHeaterShakerError`.

Latch Control

To add and remove labware from the Heater-Shaker, control the module's labware latch from your protocol using **`open_labware_latch()`** and **`close_labware_latch()`**. Shaking requires the labware latch to be closed, so you may want to issue a close command before the first shake command in your protocol:

```
hs_mod.close_labware_latch()
hs_mod.set_and_wait_for_shake_speed(500)
```

If the labware latch is already closed, `close_labware_latch()` will succeed immediately; you don't have to check the status of the latch before opening or closing it.

To prepare the deck before running a protocol, use the labware latch controls in the Opentrons App or run these methods in Jupyter notebook.

Loading Labware

Use the Heater-Shaker's **`load_adapter()`** and **`load_labware()`** methods to specify what you will place on the module. For the Heater-Shaker, use one of the thermal adapters listed below and labware that fits on the adapter. See Loading Labware on Adapters for examples of loading labware on modules.

The Opentrons Labware Library includes definitions for both standalone adapters and adapter–labware combinations. These labware definitions help make the Heater-Shaker ready to use right out of the box.

> **Note:**
>
> If you plan to move labware onto or off of the Heater-Shaker during your protocol, you must use a standalone adapter definition, not an adapter–labware combination definiton.

Standalone Adapters

You can use these standalone adapter definitions to load Opentrons verified or custom labware on top of the Heater-Shaker.

Adapter Type	API Load Name
Opentrons Universal Flat Heater-Shaker Adapter	`opentrons_universal_flat_adapter`
Opentrons 96 PCR Heater-Shaker Adapter	`opentrons_96_pcr_adapter`
Opentrons 96 Deep Well Heater-Shaker Adapter	`opentrons_96_deep_well_adapter`
Opentrons 96 Flat Bottom Heater-Shaker Adapter	`opentrons_96_flat_bottom_adapter`

For example, these commands load a well plate on top of the flat bottom adapter:

```
hs_adapter = hs_mod.load_adapter('opentrons_96_flat_bottom_adapter')
hs_plate = hs_adapter.load_labware('nest_96_wellplate_200ul_flat')
```

New in version 2.15: The `load_adapter()` method.

col specifies an `apiLevel` of 2.15 or higher, you should use the standalone adapter definitions instead.

Adapter/Labware Combination	API Load Name
Opentrons 96 Deep Well Adapter with NEST Deep Well Plate 2 mL	`opentrons_96_deep_well_adapter_nest_wellplate_2ml_deep`
Opentrons 96 Flat Bottom Adapter with NEST 96 Well Plate 200 µL Flat	`opentrons_96_flat_bottom_adapter_nest_wellplate_200ul_flat`
Opentrons 96 PCR Adapter with Armadillo Well Plate 200 µL	`opentrons_96_pcr_adapter_armadillo_wellplate_200ul`
Opentrons 96 PCR Adapter with NEST Well Plate 100 µL	`opentrons_96_pcr_adapter_nest_wellplate_100ul_pcr_full_skirt`
Opentrons Universal Flat Adapter with Corning 384 Well Plate 112 µL Flat	`opentrons_universal_flat_adapter_corning_384_wellplate_112ul_flat`

This command loads the same physical adapter and labware as the example in the previous section, but it is also compatible with API versions 2.13 and 2.14:

```
hs_combo = hs_mod.load_labware(
    "opentrons_96_flat_bottom_adapter_nest_wellplate_200ul_flat"
)
```

New in version 2.13.

Custom Flat-Bottom Labware

Custom flat-bottom labware can be used with the Universal Flat Adapter. See the support article Requesting a Custom Labware Definition if you need assistance creating custom labware definitions for the Heater-Shaker.

Heating and Shaking

The API treats heating and shaking as separate, independent activities due to the amount of time they take.

Increasing or reducing shaking speed takes a few seconds, so the API treats these actions as *blocking* commands. All other commands cannot run until the module reaches the required speed.

Heating the module, or letting it passively cool, takes more time than changing the shaking speed. As a result, the API gives you the flexibility to perform other pipetting actions while waiting for the module to reach a target temperature. When holding at temperature, you can design your protocol to run in a blocking or non-blocking manner.

> **Note:**
>
> Since API version 2.13, only the Heater-Shaker Module supports non-blocking command execution. All other modules' methods are blocking commands.

Blocking commands

This example uses a blocking command and shakes a sample for one minute. No other commands will execute until a minute has elapsed. The three commands in this example start the shake, wait for one minute, and then stop the shake:

```
hs_mod.set_and_wait_for_shake_speed(500)
protocol.delay(minutes=1)
hs_mod.deactivate_shaker()
```

```
hs_mod.set_and_wait_for_temperature(75)
protocol.delay(minutes=1)
hs_mod.deactivate_heater()
```

This may take much longer, depending on the thermal block used, the volume and type of liquid contained in the labware, and the initial temperature of the module.

Non-blocking commands

To pipette while the Heater-Shaker is heating, use `set_target_temperature()` and `wait_for_temperature()` instead of `set_and_wait_for_temperature()`:

```
hs_mod.set_target_temperature(75)
pipette.pick_up_tip()
pipette.aspirate(50, plate['A1'])
pipette.dispense(50, plate['B1'])
pipette.drop_tip()
hs_mod.wait_for_temperature()
protocol.delay(minutes=1)
hs_mod.deactivate_heater()
```

This example would likely take just as long as the blocking version above; it's unlikely that one aspirate and one dispense action would take longer than the time for the module to heat. However, be careful when putting a lot of commands between a `set_target_temperature()` call and a `delay()` call. In this situation, you're relying on `wait_for_temperature()` to resume execution of commands once heating is complete. But if the temperature has already been reached, the delay will begin later than expected and the Heater-Shaker will hold at its target temperature longer than intended.

Additionally, if you want to pipette while the module holds a temperature for a certain length of time, you need to track the holding time yourself. One of the simplest ways to do this is with Python's `time` module. First, add `import time` at the start of your protocol. Then, use `time.monotonic()` to set a reference time when the target is reached. Finally, add a delay that calculates how much holding time is remaining after the pipetting actions:

```
hs_mod.set_and_wait_for_temperature(75)
start_time = time.monotonic()  # set reference time
pipette.pick_up_tip()
pipette.aspirate(50, plate['A1'])
pipette.dispense(50, plate['B1'])
pipette.drop_tip()
# delay for the difference between now and 60 seconds after the reference time
protocol.delay(max(0, start_time+60 - time.monotonic()))
hs_mod.deactivate_heater()
```

Provided that the parallel pipetting actions don't take more than one minute, this code will deactivate the heater one minute after its target was reached. If more than one minute has elapsed, the value passed to `protocol.delay()` will equal 0, and the protocol will continue immediately.

Deactivating

Deactivating the heater and shaker are done separately using the `deactivate_heater()` and `deactivate_shaker()` methods, respectively. There is no method to deactivate both simultaneously. Call the two methods in sequence if you need to stop both heating and shaking.

> **Note:**
>
> The robot will not automatically deactivate the Heater-Shaker at the end of a protocol. If you need to deactivate the module after a protocol is completed or canceled, use the Heater-Shaker module controls on the device detail page in the Opentrons App or run these methods in Jupyter notebook.

opentrons

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Pipettes

When writing a protocol, you must inform the Protocol API about the pipettes you will be using on your robot. The `ProtocolContext.load_instrument()` function provides this information and returns an `InstrumentContext` object.

For information about liquid handling, see Building Block Commands and Complex Commands.

Loading Pipettes

As noted above, you call the `load_instrument()` method to load a pipette. This method also requires the pipette's API load name, its left or right mount position, and (optionally) a list of associated tip racks. Even if you don't use the pipette anywhere else in your protocol, the Opentrons App and the robot won't let you start the protocol run until all pipettes loaded by `load_instrument()` are attached properly.

Loading Flex 1- and 8-Channel Pipettes

This code sample loads a Flex 1-Channel Pipette in the left mount and a Flex 8-Channel Pipette in the right mount. Both pipettes are 1000 µL. Each pipette uses its own 1000 µL tip rack.

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel':'2.15'}

def run(protocol: protocol_api.ProtocolContext):
    tiprack1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_1000ul', location='D1')
    tiprack2 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_1000ul', location='C1')
    left = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack1])
    right = protocol.load_instrument(
        instrument_name='flex_8channel_1000',
        mount='right',
        tip_racks=[tiprack2])
```

If you're writing a protocol that uses the Flex Gripper, you might think that this would be the place in your protocol to declare that. However, the gripper doesn't require `load_instrument`! Whether your gripper requires a protocol is determined by the presence of `ProtocolContext.move_labware()` commands. See Moving Labware for more details.

Loading a Flex 96-Channel Pipette

This code sample loads the Flex 96-Channel Pipette. Because of its size, the Flex 96-Channel Pipette requires the left *and* right pipette mounts. You cannot use this pipette with 1- or 8-Channel Pipette in the same protocol or when these instruments are attached to the robot. To load the 96-Channel Pipette, specify its position as `mount='left'` as shown here:

```
def run(protocol: protocol_api.ProtocolContext):
    left = protocol.load_instrument(
        instrument_name='flex_96channel_1000', mount='left')
```

New in version 2.15.

👋 Hi there! What brings you to Opentrons today?

the right mount. Each pipette uses its own 1000 µL tip rack.

```
from opentrons import protocol_api

metadata = {'apiLevel': '2.14'}

def run(protocol: protocol_api.ProtocolContext):
    tiprack1 = protocol.load_labware(
        load_name='opentrons_96_tiprack_1000ul', location=1)
    tiprack2 = protocol.load_labware(
        load_name='opentrons_96_tiprack_1000ul', location=2)
    left = protocol.load_instrument(
        instrument_name='p1000_single_gen2',
        mount='left',
        tip_racks=[tiprack1])
    right = protocol.load_instrument(
        instrument_name='p300_multi_gen2',
        mount='right',
        tip_racks=[tiprack1])
```

New in version 2.0.

Multi-Channel Pipettes

All building block and advanced commands work with single- and multi-channel 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.

Also, this means that offset changes (such as `Well.top()` or `Well.bottom()`) can be applied to the single specified well, and each pipette channel will be at the same position relative to the well that it is over.

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

8-Channel, 96-Well Plate Example

To demonstrate these concepts, let's write a protocol that uses a Flex 8-Channel Pipette and a 96-well plate. We'll then aspirate and dispense a liquid to different locations on the same well plate. To start, let's load a pipette in the right mount and add our labware.

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel':'2.15'}

def run(protocol: protocol_api.ProtocolContext):
    # Load a tiprack for 1000 µL tips
    tiprack1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_1000ul', location='D1')
    # Load a 96-well plate
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat', location='C1')
    # Load an 8-channel pipette on the right mount
    right = protocol.load_instrument(
        instrument_name='flex_8channel_1000',
        mount='right',
        tip_racks=[tiprack1])
```

After loading our instruments and labware, let's tell the robot to pick up a pipette tip from location `A1` in `tiprack1`:

```
right.pick_up_tip()
```

After picking up a tip, let's tell the robot to aspirate 300 µL from the well plate at location `A2`:

```
right.aspirate(volume=300, location=plate['A2'])
```

With the backmost pipette tip above location A2 on the well plate, all eight channels are above the eight wells in column 2.

Finally, let's tell the robot to dispense 300 µL into the well plate at location `A3`:

```
right.dispense(volume=300, location=plate['A3'].top())
```

With the backmost pipette tip above location A3, all eight channels are above the eight wells in column 3. The pipette will dispense liquid into all the wells simultaneously.

8-Channel, 384-Well Plate Example

In general, you should specify wells in the first row of a well plate when using multi-channel pipettes. An exception to this rule is when using 384-well plates. The greater well density means the nozzles of a multi-channel pipette can only access every other well in a column. Specifying well A1 accesses every other well starting with the first (rows A, C, E, G, I, K, M, and O). Similarly, specifying well B1 also accesses every other well, but starts with the second (rows B, D, F, H, J, L, N, and P).

To demonstrate these concepts, let's write a protocol that uses a Flex 8-Channel Pipette and a 384-well plate. We'll then aspirate and dispense a liquid to different locations on the same well plate. To start, let's load a pipette in the right mount and add our labware.

```
def run(protocol: protocol_api.ProtocolContext):
    # Load a tiprack for 200 µL tips
    tiprack1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul', location="D1")
    # Load a well plate
    plate = protocol.load_labware(
        load_name='corning_384_wellplate_112ul_flat', location="D2")
    # Load an 8-channel pipette on the right mount
    right = protocol.load_instrument(
        instrument_name='flex_8channel_1000',
        mount='right',
        tip_racks=[tiprack1])
```

After loading our instruments and labware, let's tell the robot to pick up a pipette tip from location `A1` in `tiprack1`:

```
right.pick_up_tip()
```

With the backmost pipette channel above location A1 on the tip rack, all eight channels are above the eight tip rack wells in column 1.

After picking up a tip, let's tell the robot to aspirate 100 µL from the well plate at location `A1`:

```
right.aspirate(volume=100, location=plate['A1'])
```

The eight pipette channels will only aspirate from every other well in the column: A1, C1, E1, G1, I1, K1, M1, and O1.

Finally, let's tell the robot to dispense 100 µL into the well plate at location `B1`:

```
right.dispense(volume=100, location=plate['B1'])
```

The eight pipette channels will only dispense into every other well in the column: B1, D1, F1, H1, J1, L1, N1, and P1.

Adding Tip Racks

specify tip locations in the `InstrumentContext.pick_up_tip()` method. For example, let's start by loading loading some labware and instruments like this:

```
def run(protocol: protocol_api.ProtocolContext):
    tiprack_left = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul', location='D1')
    tiprack_right = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul', location='D2')
    left_pipette = protocol.load_instrument(
        instrument_name='flex_8channel_1000', mount='left')
    right_pipette = protocol.load_instrument(
        instrument_name='flex_8channel_1000',
        mount='right',
        tip_racks=[tiprack_right])
```

Let's pick up a tip with the left pipette. We need to specify the location as an argument of `pick_up_tip()`, since we loaded the left pipette without a `tip_racks` argument.

```
left_pipette.pick_up_tip(tiprack_left['A1'])
left_pipette.drop_tip()
```

But now you have to specify `tiprack_left` every time you call `pick_up_tip`, which means you're doing all your own tip tracking:

```
left_pipette.pick_up_tip(tiprack_left['A2'])
left_pipette.drop_tip()
left_pipette.pick_up_tip(tiprack_left['A3'])
left_pipette.drop_tip()
```

However, because you specified a tip rack location for the right pipette, the robot will automatically pick up from location `A1` of its associated tiprack:

```
right_pipette.pick_up_tip()
right_pipette.drop_tip()
```

Additional calls to `pick_up_tip` will automatically progress through the tips in the right rack:

```
right_pipette.pick_up_tip()  # picks up from A2
right_pipette.drop_tip()
right_pipette.pick_up_tip()  # picks up from A3
right_pipette.drop_tip()
```

See also, Building Block Commands and Complex Commands.

New in version 2.0.

API Load Names

The pipette's API load name (`instrument_name`) is the first parameter of the `load_instrument()` method. It tells your robot which attached pipette you're going to use in a protocol. The tables below list the API load names for the currently available Flex and OT-2 pipettes.

Flex Pipettes | **OT-2 Pipettes**

Pipette Model	Volume (µL)	API Load Name
P20 Single-Channel GEN2	1-20	`p20_single_gen2`
P20 Multi-Channel GEN2		`p20_multi_gen2`

P300 Multi-Channel GEN2		`p300_multi_gen2`
P1000 Single-Channel GEN2	100-1000	`p1000_single_gen2`

See the OT-2 Pipette Generations section below if you're using GEN1 pipettes on an OT-2. The GEN1 family includes the P10, P50, and P300 single- and multi-channel pipettes, along with the P1000 single-chanel model.

OT-2 Pipette Generations

The OT-2 works with the GEN1 and GEN2 pipette models. The newer GEN2 pipettes have different volume ranges than the older GEN1 pipettes. With some exceptions, the volume ranges for GEN2 pipettes overlap those used by the GEN1 models. If your protocol specifies a GEN1 pipette, but you have a GEN2 pipette 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. The following table lists the volume compatibility between the GEN2 and GEN1 pipettes.

GEN2 Pipette	GEN1 Pipette	GEN1 Volume
P20 Single-Channel GEN2	P10 Single-Channel GEN1	1-10 µL
P20 Multi-Channel GEN2	P10 Multi-Channel GEN1	1-10 µL
P300 Single-Channel GEN2	P300 Single-Channel GEN1	30-300 µL
P300 Multi-Channel GEN2	P300 Multi-Channel GEN1	20-200 µL
P1000 Single-Channel GEN2	P1000 Single-Channel GEN1	100-1000 µL

The single- and multi-channel P50 GEN1 pipettes are the exceptions here. If your protocol uses a P50 GEN1 pipette, there is no backward compatibility with a related GEN2 pipette. To replace a P50 GEN1 with a corresponding GEN2 pipette, edit your protocol to load a P20 Single-Channel GEN2 (for volumes below 20 µL) or a P300 Single-Channel GEN2 (for volumes between 20 and 50 µL).

Volume Modes

The Flex 1-Channel 50 µL and Flex 8-Channel 50 µL pipettes must operate in a low-volume mode to accurately dispense very small volumes of liquid. Set the volume mode by calling `InstrumentContext.configure_for_volume()` with the amount of liquid you plan to aspirate, in µL:

```
pipette50.configure_for_volume(1)
pipette50.pick_up_tip()
pipette50.aspirate(1, plate["A1"])
```

New in version 2.15.

Passing different values to `configure_for_volume()` changes the minimum and maximum volume of Flex 50 µL pipettes as follows:

Value	Minimum Volume (µL)	Maximum Volume (µL)
1–4.9	1	30
5–50	5	50

Note:

The pipette must not contain liquid when you call `configure_for_volume()`, or the API will raise an error.

In a protocol that handles many different volumes, it's a good practice to call `configure_for_volume()` once for each **`transfer()`** or **`aspirate()`**, specifying the volume that you are about to handle. When operating with a list of volumes, nest `configure_for_volume()` inside a `for` loop to ensure that the pipette is properly configured for each volume:

```
volumes = [1, 2, 3, 4, 1, 5, 2, 8]
sources = plate.columns()[0]
destinations = plate.columns()[1]
for i in range(8):
    pipette50.configure_for_volume(volumes[i])
    pipette50.pick_up_tip()
    pipette50.aspirate(volume=volumes[i], location=sources[i])
    pipette50.dispense(location=destinations[i])
    pipette50.drop_tip()
```

If you know that all your liquid handling will take place in a specific mode, then you can call `configure_for_volume()` just once with a representative volume. Or if all the volumes correspond to the pipette's default mode, you don't have to call `configure_for_volume()` at all.

Pipette Flow Rates

Measured in µL/s, the flow rate determines how much liquid a pipette can aspirate, dispense, and blow out. Opentrons pipettes have their own default flow rates. The API lets you change the flow rate on a loaded **`InstrumentContext`** by altering the **`InstrumentContext.flow_rate`** properties listed below.

- Aspirate: `InstrumentContext.flow_rate.aspirate`
- Dispense: `InstrumentContext.flow_rate.dispense`
- Blow out: `InstrumentContext.flow_rate.blow_out`

These flow rate properties operate independently. This means you can specify different flow rates for each property within the same protocol. For example, let's load a simple protocol and set different flow rates for the attached pipette.

```
def run(protocol: protocol_api.ProtocolContext):
    tiprack1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_1000ul', location='D1')
    pipette = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack1])
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat', location='D3')
    pipette.pick_up_tip()
```

Let's tell the robot to aspirate, dispense, and blow out the liquid using default flow rates. Notice how you don't need to specify a `flow_rate` attribute to use the defaults:

```
pipette.aspirate(200, plate['A1'])  # 160 µL/s
pipette.dispense(200, plate['A2'])  # 160 µL/s
pipette.blow_out()                  #  80 µL/s
```

Now let's change the flow rates for each action:

```
pipette.flow_rate.aspirate = 50
pipette.flow_rate.dispense = 100
pipette.flow_rate.blow_out = 75
pipette.aspirate(200, plate['A1'])  #  50 µL/s
pipette.dispense(200, plate['A2'])  # 100 µL/s
pipette.blow_out()                  #  75 µL/s
```

These flow rates will remain in effect until you change the `flow_rate` attribute again *or* call `configure_for_volume()`. Calling `configure_for_volume()` always resets all pipette flow rates to the defaults for the mode that it sets.

set the plunger speed in mm/s. Due to technical limitations, that speed could only be approximate. You must use `.flow_rate` in version 2.14 and later, and you should consider replacing older code that sets `.speed`.

New in version 2.0.

Flex Pipette Flow Rates

The default flow rates for Flex pipettes depend on the maximum volume of the pipette and the capacity of the currently attached tip. For each pipette–tip configuration, the default flow rate is the same for aspirate, dispense, and blowout actions.

Pipette Model	Tip Capacity (µL)	Flow Rate (µL/s)
50 µL (1- and 8-channel)	All capacities	57
1000 µL (1-, 8-, and 96-channel)	50	478
1000 µL (1-, 8-, and 96-channel)	200	716
1000 µL (1-, 8-, and 96-channel)	1000	716

Additionally, all Flex pipettes have a well bottom clearance of 1 mm for aspirate and dispense actions.

OT-2 Pipette Flow Rates

The following table provides data on the default aspirate, dispense, and blowout flow rates (in µL/s) for OT-2 GEN2 pipettes. Default flow rates are the same across all three actions.

Pipette Model	Volume (µL)	Flow Rates (µL/s)
P20 Single-Channel GEN2	1–20	• API v2.6 or higher: 7.56 • API v2.5 or lower: 3.78
P300 Single-Channel GEN2	20–300	• API v2.6 or higher: 92.86 • API v2.5 or lower: 46.43
P1000 Single-Channel GEN2	100–1000	• API v2.6 or higher: 274.7 • API v2.5 or lower: 137.35
P20 Multi-Channel GEN2	1–20	7.6
P300 Multi-Channel GEN2	20–300	94

Additionally, all OT-2 GEN2 pipettes have a default head speed of 400 mm/s and a well bottom clearance of 1 mm for aspirate and dispense actions.

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Sources and Destinations

The `InstrumentContext.transfer()`, `InstrumentContext.distribute()`, and `InstrumentContext.consolidate()` methods form the family of complex liquid handling commands. These methods require `source` and `dest` (destination) arguments to move liquid from one well, or group of wells, to another. In contrast, the building block commands `aspirate()` and `dispense()` only operate in a single location.

For example, this command performs a simple transfer between two wells on a plate:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=plate["A2"],
)
```

New in version 2.0.

This page covers the restrictions on sources and destinations for complex commands, their different patterns of aspirating and dispensing, and how to optimize them for different use cases.

Source and Destination Arguments

As noted above, the `transfer()`, `distribute()`, and `consolidate()` methods require `source` and `dest` (destination) arguments to aspirate and dispense liquid. However, each method handles liquid sources and destinations differently. Understanding how complex commands work with source and destination wells is essential to using these methods effectively.

`transfer()` is the most versatile complex liquid handling function, because it has the fewest restrictions on what wells it can operate on. You will likely use transfer commands in many of your protocols.

Certain liquid handling cases focus on moving liquid to or from a single well. `distribute()` limits its source to a single well, while `consolidate()` limits its destination to a single well. Distribute commands also make changes to liquid-handling behavior to improve the accuracy of dispensing.

The following table summarizes the source and destination restrictions for each method.

Method	Accepted wells
`transfer()`	• **Source:** Any number of wells. • **Destination:** Any number of wells. • The larger group of wells must be evenly divisible by the smaller group.
`distribute()`	• **Source:** Exactly one well. • **Destination:** Any number of wells.
`consolidate()`	• **Source:** Any number of wells. • **Destination:** Exactly one well.

A single well can be passed by itself or as a list with one item: `source=plate['A1']` and `source=[plate['A1']]` are equivalent.

The section on many-to-many transfers below covers how `transfer()` works when specifying sources and destinations of different sizes. However, if they don't meet the even divisibility requirement, the API will raise an error. You can work

For distributing and consolidating, the API will not raise an error if you use a list of wells as the argument that is limited to exactly one well. Instead, the API will ignore everything except the first well in the list. For example, the following command will only aspirate from well A1:

```
pipette.distribute(
    volume=100,
    source=[plate["A1"], plate["A2"]],  # A2 ignored
    dest=plate.columns()[1],
)
```

On the other hand, a transfer command with the same arguments would aspirate from both A1 and A2. The next section examines the exact order of aspiration and dispensing for all three methods.

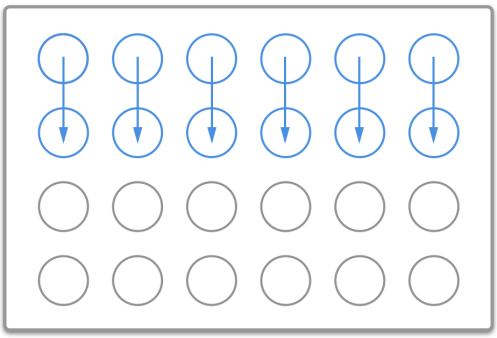
Transfer Patterns

Each complex command uses a different pattern of aspiration and dispensing. In addition, when you provide multiple wells as both the source and destination for `transfer()`, it maps the source list onto the destination list in a certain way.

Aspirating and Dispensing

`transfer()` always alternates between aspirating and dispensing, regardless of how many wells are in the source and destination. Its default behavior is:

1. Pick up a tip.
2. Aspirate from the first source well.
3. Dispense in the first destination well.
4. Repeat the pattern of aspirating and dispensing, as needed.
5. Drop the tip in the trash.

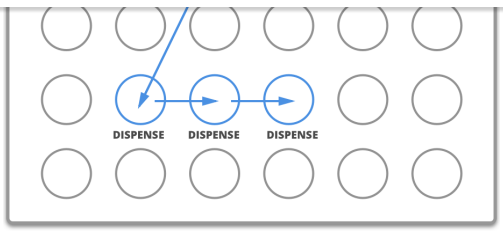


This transfer aspirates six times and dispenses six times.

`distribute()` always fills the tip with as few aspirations as possible, and then dispenses to the destination wells in order. Its default behavior is:

1. Pick up a tip.
2. Aspirate enough to dispense in all the destination wells. This aspirate includes a disposal volume.
3. Dispense in the first destination well.
4. Continue to dispense in destination wells.
5. Drop the tip in the trash.

See Tip Refilling below for cases where the total amount to be dispensed is greater than the capacity of the tip.

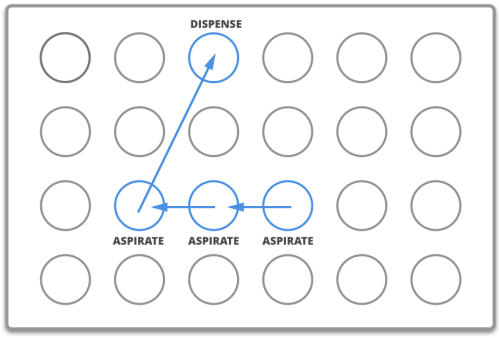


This distribute aspirates one time and dispenses three times.

`consolidate()` aspirates multiple times in a row, and then dispenses as few times as possible in the destination well. Its default behavior is:

1. Pick up a tip.
2. Aspirate from the first source well.
3. Continue aspirating from source wells.
4. Dispense in the destination well.
5. Drop the tip in the trash.

See Tip Refilling below for cases where the total amount to be aspirated is greater than the capacity of the tip.



This consolidate aspirates three times and dispenses one time.

Note:

By default, all three commands begin by picking up a tip and conclude by dropping a tip. In general, don't call `pick_up_tip()` just before a complex command, or the API will raise an error. You can override this behavior with the tip handling complex parameter, by setting `new_tip="never"`.

Many-to-Many

`transfer()` lets you specify both `source` and `dest` arguments that contain multiple wells. This section covers how the method determines which wells to aspirate from and dispense to in these cases.

When the source and destination both contain the same number of wells, the mapping between wells is straightforward. You can imagine writing out the two lists one above each other, with each unique well in the source list paired to a unique well in the destination list. For example, here is the code for using one row as the source and another row as the destination, and the resulting correspondence between wells:

```
pipette.transfer(
    volume=50,
    source=plate.rows()[0],
```

Destination	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12

There's no requirement that the source and destination lists be mutually exclusive. In fact, this command adapted from the tutorial deliberately uses slices of the same list, saved to the variable `row`, with the effect that each aspiration happens in the same location as the previous dispense:

```
row = plate.rows()[0]
pipette.transfer(
    volume=50,
    source=row[:11],
    dest=row[1:],
)
```

Source	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
Destination	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12

When the source and destination lists contain different numbers of wells, `transfer()` will always aspirate and dispense as many times as there are wells in the *longer* list. The shorter list will be "stretched" to cover the length of the longer list. Here is an example of transferring from 3 wells to a full row of 12 wells:

```
pipette.transfer(
    volume=50,
    source=[plate["A1"], plate["A2"], plate["A3"]],
    dest=plate.rows()[1],
)
```

Source	A1	A1	A1	A1	A2	A2	A2	A2	A3	A3	A3	A3
Destination	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12

This is why the longer list must be evenly divisible by the shorter list. Changing the destination in this example to a column instead of a row will cause the API to raise an error, because 8 is not evenly divisible by 3:

```
pipette.transfer(
    volume=50,
    source=[plate["A1"], plate["A2"], plate["A3"]],
    dest=plate.columns()[3],  # labware column 4
)
# error: source and destination lists must be divisible
```

The API raises this error rather than presuming which wells to aspirate from three times and which only two times. If you want to aspirate three times from A1, three times from A2, and two times from A3, use multiple `transfer()` commands in sequence:

```
pipette.transfer(50, plate["A1"], plate.columns()[3][:3])
pipette.transfer(50, plate["A2"], plate.columns()[3][3:6])
pipette.transfer(50, plate["A3"], plate.columns()[3][6:])
```

Finally, be aware of the ordering of source and destination lists when constructing them with well accessor methods. For example, at first glance this code may appear to take liquid from each well in the first row of a plate and move it to each of the other wells in the same column:

```
pipette.transfer(
    volume=20,
    source=plate.rows()[0],
    dest=plate.rows()[1:],
)
```

However, because the well ordering of `Labware.rows()` goes *across* the plate instead of *down* the plate, liquid from A1 will be dispensed in B1–B7, liquid from A2 will be dispensed in B8–C2, etc. The intended task is probably better accomplished

```
            volume=20,
            source=plate.rows()[0][i],
            dest=plate.columns()[i][1:],
        )
```

Here the repeat index `i` picks out:

- The individual well in the first row, for the source.
- The corresponding column, which is sliced to form the destination.

Optimizing Patterns

Choosing the right complex command optimizes gantry movement and helps save time in your protocol. For example, say you want to take liquid from a reservoir and put 50 µL in each well of the first row of a plate. You could use `transfer()`, like this:

```
pipette.transfer(
    volume=50,
    source=reservoir["A1"],
    destination=plate.rows()[0],
)
```

This will produce 12 aspirate steps and 12 dispense steps. The steps alternate, with the pipette moving back and forth between the reservoir and plate each time. Using `distribute()` with the same arguments is more optimal in this scenario:

```
pipette.distribute(
    volume=50,
    source=reservoir["A1"],
    destination=plate.rows()[0],
)
```

This will produce *just 1* aspirate step and 12 dispense steps (when using a 1000 µL pipette). The pipette will aspirate enough liquid to fill all the wells, plus a disposal volume. Then it will move to A1 of the plate, dispense, move the short distance to A2, dispense, and so on. This greatly reduces gantry movement and the time to perform this action. And even if you're using a smaller pipette, `distribute()` will fill the pipette, dispense as many times as possible, and only then return to the reservoir to refill (see Tip Refilling for more information).

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Order of Operations

Complex commands perform a series of building block commands in order. In fact, the run preview for your protocol in the Opentrons App lists all of these commands as separate steps. This lets you examine what effect your complex commands will have before running them.

This page describes what steps you should expect the robot to perform when using different complex commands with different required and optional parameters.

Step Sequence

The order of steps is fixed within complex commands. Aspiration and dispensing are the only required actions. You can enable or disable all of the other actions with complex liquid handling parameters. A complex command designed to perform every possible action will proceed in this order:

1. Pick up tip
2. Mix at source
3. Aspirate from source
4. Touch tip at source
5. Air gap
6. Dispense into destination
7. Mix at destination
8. Touch tip at destination
9. Blow out
10. Drop tip

The command may repeat some or all of these steps in order to move liquid as requested. `transfer()` repeats as many times as there are wells in the longer of its `source` or `dest` argument. `distribute()` and `consolidate()` try to repeat as few times as possible. See Tip Refilling below for how they behave when they do need to repeat.

Example Orders

The smallest possible number of steps in a complex command is just two: aspirating and dispensing. This is possible by omitting the tip pickup and drop steps:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=plate["B1"],
    new_tip="never",
)
```

Here's another example, a distribute command that adds touch tip steps (and does not turn off tip handling). The code for this command is:

```
pipette.distribute(
    volume=100,
    source=[plate["A1"]],
    dest=[plate["B1"], plate["B2"]],
```

1. Pick up tip
2. Aspirate from source
3. Touch tip at source
4. Dispense into destination
5. Touch tip at destination
6. Blow out
7. Drop tip

Let's unpack this. Picking up and dropping tips is default behavior for `distribute()`. Specifying `touch_tip=True` adds two steps, as it is performed at both the source and destination. And it's also default behavior for `distribute()` to aspirate a disposal volume, which is blown out before dropping the tip. The exact order of steps in the run preview should look similar to this:

```
Picking up tip from A1 of tip rack on 3
Aspirating 220.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Touching tip
Dispensing 100.0 uL into B1 of well plate on 2 at 92.86 uL/sec
Touching tip
Dispensing 100.0 uL into B2 of well plate on 2 at 92.86 uL/sec
Touching tip
Blowing out at A1 of Opentrons Fixed Trash on 12
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

Since dispensing and touching the tip are both associated with the destination wells, those steps are performed at each of the two destination wells.

Tip Refilling

One factor that affects the exact order of steps for a complex command is whether the amount of liquid being moved can fit in the tip at once. If it won't fit, you don't have to adjust your command. The API will handle it for you by including additional steps to refill the tip when needed.

For example, say you need to move 100 µL of liquid from one well to another, but you only have a 50 µL pipette attached to your robot. To accomplish this with building block commands, you'd need multiple aspirates and dispenses. `aspirate(volume=100)` would raise an error, since it exceeds the tip's volume. But you can accomplish this with a single transfer command:

```
pipette50.transfer(
    volume=100,
    source=plate["A1"],
    dest=plate["B1"],
)
```

To effect the transfer, the API will aspirate and dispense the maximum volume of the pipette (50 µL) twice:

```
Picking up tip from A1 of tip rack on D3
Aspirating 50.0 uL from A1 of well plate on D2 at 57 uL/sec
Dispensing 50.0 uL into B1 of well plate on D2 at 57 uL/sec
Aspirating 50.0 uL from A1 of well plate on D2 at 57 uL/sec
Dispensing 50.0 uL into B1 of well plate on D2 at 57 uL/sec
Dropping tip into A1 of Opentrons Fixed Trash on A3
```

You can change `volume` to any value (above the minimum volume of the pipette) and the API will automatically calculate how many times the pipette needs to aspirate and dispense. `volume=50` would require just one repetition. `volume=75` would require two, split into 50 µL and 25 µL. `volume=1000` would repeat 20 times — not very efficient, but perhaps more useful than having to swap to a different pipette!

```
Picking up tip from A1 of tip rack on 3
Aspirating 980.0 uL from A1 of well plate on 2 at 274.7 uL/sec
Dispensing 80.0 uL into B1 of well plate on 2 at 274.7 uL/sec
Dispensing 80.0 uL into B2 of well plate on 2 at 274.7 uL/sec
...
Dispensing 80.0 uL into B11 of well plate on 2 at 274.7 uL/sec
Blowing out at A1 of Opentrons Fixed Trash on 12
Aspirating 180.0 uL from A1 of well plate on 2 at 274.7 uL/sec
Dispensing 80.0 uL into B12 of well plate on 2 at 274.7 uL/sec
Blowing out at A1 of Opentrons Fixed Trash on 12
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

This command will blow out 200 total µL of liquid in the trash. If you need to conserve liquid, use complex liquid handling parameters to reduce or eliminate the disposal volume, or to blow out in a location other than the trash.

List of Volumes

Complex commands can aspirate or dispense different amounts for different wells, rather than the same amount across all wells. To do this, set the `volume` parameter to a list of volumes instead of a single number. The list must be the same length as the longer of `source` or `dest`, or the API will raise an error. For example, this command transfers a different amount of liquid into each of wells B1, B2, and B3:

```
pipette.transfer(
    volume=[20, 40, 60],
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"], plate["B3"]],
)
```

Setting any item in the list to `0` will skip aspirating and dispensing for the corresponding well. This example takes the command from above and skips B2:

```
pipette.transfer(
    volume=[20, 0, 60],
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"], plate["B3"]],
)
```

The pipette dispenses in B1 and B3, and does not move to B2 at all.

```
Picking up tip from A1 of tip rack on 3
Aspirating 20.0 uL from A1 of well plate on 2 at 274.7 uL/sec
Dispensing 20.0 uL into B1 of well plate on 2 at 274.7 uL/sec
Aspirating 60.0 uL from A1 of well plate on 2 at 274.7 uL/sec
Dispensing 60.0 uL into B3 of well plate on 2 at 274.7 uL/sec
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

This is such a simple example that you might prefer to use two `transfer()` commands instead. Lists of volumes become more useful when they are longer than a couple elements. For example, you can specify `volume` as a list with 96 items and `dest=plate.wells()` to individually control amounts to dispense (and wells to skip) across an entire plate.

> **Note:**
>
> When the optional `new_tip` parameter is set to `"always"`, the pipette will pick up and drop a tip even for skipped wells. If you don't want to waste tips, pre-process your list of sources or destinations and use the result as the argument of your complex command.

New in version 2.0: Skip wells for `transfer()` and `distribute()`.

New in version 2.8: Skip wells for `consolidate()`.

opentrons

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Complex Liquid Handling Parameters

Complex commands accept a number of optional parameters that give you greater control over the exact steps they perform.

This page describes the accepted values and behavior of each parameter. The parameters are organized in the order that they first add a step. Some parameters, such as `touch_tip`, add multiple steps. See Order of Operations for more details on the sequence of steps performed by complex commands.

The API reference entry for `InstrumentContext.transfer()` also lists the parameters and has more information on their implementation as keyword arguments.

Tip Handling

The `new_tip` parameter controls if and when complex commands pick up new tips from the pipette's tip racks. It has three possible values:

Value	Behavior
`"once"`	• Pick up a tip at the start of the command. • Use the tip for all liquid handling. • Drop the tip at the end of the command.
`"always"`	Pick up and drop a tip for each set of aspirate and dispense steps.
`"never"`	Do not pick up or drop tips at all.

`"once"` is the default behavior for all complex commands.

New in version 2.0.

Tip Handling Requirements

`"once"` and `"always"` require that the pipette has an associated tip rack, or the API will raise an error (because it doesn't know where to pick up a tip from). If the pipette already has a tip attached, the API will also raise an error when it tries to pick up a tip.

```
pipette.pick_up_tip()
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"], plate["B3"]],
    new_tip="never",  # "once", "always", or None will error
)
```

Conversely, `"never"` requires that the pipette has picked up a tip, or the API will raise an error (because it will attempt to aspirate without a tip attached).

Avoiding Cross-Contamination

`transfer()` will pick up a new tip before *every* aspirate when `new_tip="always"`. This includes when tip refilling requires multiple aspirations from a single source well.

`distribute()` and `consolidate()` only pick up one tip, even when `new_tip="always"`. For example, this distribute command returns to the source well a second time, because the amount to be distributed (400 µL total plus disposal volume) exceeds the pipette capacity (300 µL):

```
pipette.distribute(
    volume=200,
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"]],
    new_tip="always",
)
```

But it *does not* pick up a new tip after dispensing into B1:

```
Picking up tip from A1 of tip rack on 3
Aspirating 220.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Dispensing 200.0 uL into B1 of well plate on 2 at 92.86 uL/sec
Blowing out at A1 of Opentrons Fixed Trash on 12
Aspirating 220.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Dispensing 200.0 uL into B2 of well plate on 2 at 92.86 uL/sec
Blowing out at A1 of Opentrons Fixed Trash on 12
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

If this poses a contamination risk, you can work around it in a few ways:

- Use `transfer()` with `new_tip="always"` instead.
- Set `well_bottom_clearance` high enough that the tip doesn't contact liquid in the destination well.
- Use building block commands instead of complex commands.

Mix Before

The `mix_before` parameter controls mixing in source wells before each aspiration. Its value must be a `tuple` with two numeric values. The first value is the number of repetitions, and the second value is the amount of liquid to mix in µL.

For example, this transfer command will mix 50 µL of liquid 3 times before each of its aspirations:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"]],
    mix_before=(3, 50),
)
```

New in version 2.0.

Mixing occurs before every aspiration, including when tip refilling is required.

> **Note:**
>
> `consolidate()` ignores any value of `mix_before`. Mixing on the second and subsequent aspirations of a consolidate command would defeat its purpose: to aspirate multiple times in a row, from different wells, *before* dispensing.

Disposal Volume

The `disposal_volume` parameter controls how much extra liquid is aspirated as part of a `distribute()` command. Including a disposal volume can improve the accuracy of each dispense. The pipette blows out the disposal volume of liquid after dispensing. To skip aspirating and blowing out extra liquid, set `disposal_volume=0`.

```
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"]],
    disposal_volume=10,  # reduce from default 20 µL to 10 µL
)
```

New in version 2.0.

If the amount to aspirate plus the disposal volume exceeds the tip's capacity, `distribute()` will use a tip refilling strategy. In such cases, the pipette will aspirate and blow out the disposal volume *for each aspiration*. For example, this command will require tip refilling with a 1000 µL pipette:

```
pipette.distribute(
    volume=120,
    source=reservoir["A1"],
    dest=[plate.columns()[0]],
    disposal_volume=50,
)
```

The amount to dispense in the destination is 960 µL (120 µL for each of 8 wells in the column). Adding the 50 µL disposal volume exceeds the 1000 µL capacity of the tip. The command will be split across two aspirations, each with the full disposal volume of 50 µL. The pipette will dispose *a total of 100 µL* during the command.

> **Note:**
>
> **`transfer()`** will not aspirate additional liquid if you set `disposal_volume`. However, it will perform a very small blow out after each dispense.
>
> **`consolidate()`** ignores `disposal_volume` completely.

Touch Tip

The `touch_tip` parameter accepts a Boolean value. When `True`, a touch tip step occurs after every aspirate and dispense.

For example, this transfer command aspirates, touches the tip at the source, dispenses, and touches the tip at the destination:

```
pipette.transfer(
    volume=100,
    dest=plate["A1"],
    source=plate["B1"],
    touch_tip=True,
)
```

New in version 2.0.

Touch tip occurs after every aspiration, including when tip refilling is required.

This parameter always uses default motion behavior for touch tip. Use the touch tip building block command if you need to:

- Only touch the tip after aspirating or dispensing, but not both.
- Control the speed, radius, or height of the touch tip motion.

Air Gap

The `air_gap` parameter controls how much air to aspirate and hold in the bottom of the tip when it contains liquid. The parameter's value is the amount of air to aspirate in µL.

Method	Air-gapping behavior
`transfer()`	• Air gap after each aspiration. • Pipette is empty after dispensing.
`distribute()`	• Air gap after each aspiration. • Air gap after dispensing if the pipette isn't empty.
`consolidate()`	• Air gap after each aspiration. This may create multiple air gaps within the tip. • Pipette is empty after dispensing.

For example, this transfer command will create a 20 µL air gap after each of its aspirations. When dispensing, it will clear the air gap and dispense the full 100 µL of liquid:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=plate["B1"],
    air_gap=20,
)
```

New in version 2.0.

When consolidating, air gaps still occur after every aspiration. In this example, the tip will use 210 µL of its capacity (50 µL of liquid followed by 20 µL of air, repeated three times):

```
pipette.consolidate(
    volume=50,
    source=[plate["A1"], plate["A2"], plate["A3"]],
    dest=plate["B1"],
    air_gap=20,
)
```

```
Picking up tip from A1 of tip rack on 3
Aspirating 50.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Air gap
    Aspirating 20.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Aspirating 50.0 uL from A2 of well plate on 2 at 92.86 uL/sec
Air gap
    Aspirating 20.0 uL from A2 of well plate on 2 at 92.86 uL/sec
Aspirating 50.0 uL from A3 of well plate on 2 at 92.86 uL/sec
Air gap
    Aspirating 20.0 uL from A3 of well plate on 2 at 92.86 uL/sec
Dispensing 210.0 uL into B1 of well plate on 2 at 92.86 uL/sec
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

If adding an air gap would exceed the pipette's maximum volume, the complex command will use a tip refilling strategy. For example, this command uses a 300 µL pipette to transfer 300 µL of liquid plus an air gap:

```
pipette.transfer(
    volume=300,
    source=plate["A1"],
    dest=plate["B1"],
    air_gap=20,
)
```

As a result, the transfer is split into two aspirates of 150 µL, each with their own 20 µL air gap:

```
Picking up tip from A1 of tip rack on 3
Aspirating 150.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Air gap
        Aspirating 20.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Dispensing 170.0 uL into B1 of well plate on 2 at 92.86 uL/sec
Aspirating 150.0 uL from A1 of well plate on 2 at 92.86 uL/sec
Air gap
        Aspirating 20.0 uL from A1 of well plate on 2 at 92.86 uL/sec
```

Mix After

The `mix_after` parameter controls mixing in source wells after each dispense. Its value must be a **`tuple`** with two numeric values. The first value is the number of repetitions, and the second value is the amount of liquid to mix in µL.

For example, this transfer command will mix 50 µL of liquid 3 times after each of its dispenses:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"]],
    mix_after=(3, 50),
)
```

New in version 2.0.

> **Note:**
>
> **`distribute()`** ignores any value of `mix_after`. Mixing after dispensing would combine (and potentially contaminate) the remaining source liquid with liquid present at the destination.

Blow Out

There are two parameters that control whether and where the pipette blows out liquid. The `blow_out` parameter accepts a Boolean value. When `True`, the pipette blows out remaining liquid when the tip is empty or only contains the disposal volume. The `blowout_location` parameter controls in which of three locations these blowout actions occur. The default blowout location is the trash. Blowout behavior is different for each complex command.

Method	Blowout behavior and location
`transfer()`	• Blow out after each dispense. • Valid locations: `"trash"`, `"source well"`, `"destination well"`
`distribute()`	• Blow out after the final dispense. • Valid locations: `"trash"`, `"source well"`
`consolidate()`	• Blow out after the only dispense. • Valid locations: `"trash"`, `"destination well"`

For example, this transfer command will blow out liquid in the trash twice, once after each dispense into a destination well:

```
pipette.transfer(
    volume=100,
    source=[plate["A1"], plate["A2"]],
    dest=[plate["B1"], plate["B2"]],
    blow_out=True,
)
```

New in version 2.0.

Set `blowout_location` when you don't want to waste any liquid by blowing it out into the trash. For example, you may want to make sure that every last bit of a sample is moved into a destination well. Or you may want to return every last bit of an expensive reagent to the source for use in later pipetting.

If you need to blow out in a different well, or at a specific location within a well, use the blow out building block command instead.

When setting a blowout location, you *must* also set `blow_out=True`, or the location will be ignored:

```
    blow_out=True,  # required to set location
    blowout_location="destination well",
)
```

New in version 2.8.

With `transfer()`, the pipette will not blow out at all if you only set `blowout_location`.

`blow_out=True` is also required for distribute commands that blow out by virtue of having a disposal volume:

```
pipette.distribute(
    volume=100,
    source=plate["A1"],
    dest=[plate["B1"], plate["B2"]],
    disposal_volume=50,  # causes blow out
    blow_out=True,       # still required to set location!
    blowout_location="source well",
)
```

With `distribute()`, the pipette will still blow out if you only set `blowout_location`, but in the default location of the trash.

Note:

If the tip already contains liquid before the complex command, the default blowout location will shift away from the trash. `transfer()` and `distribute()` shift to the source well, and `consolidate()` shifts to the destination well. For example, this transfer command will blow out in well B1 because it's the source:

```
pipette.pick_up_tip()
pipette.aspirate(100, plate["A1"])
pipette.transfer(
    volume=100,
    source=plate["B1"],
    dest=plate["C1"],
    new_tip="never",
    blow_out=True,
    # no blowout_location
)
pipette.drop_tip()
```

This only occurs when you aspirate and then perform a complex command with `new_tip="never"` and `blow_out=True`.

Trash Tips

The `trash` parameter controls what the pipette does with tips at the end of complex commands. When `True`, the pipette drops tips into the trash. When `False`, the pipette returns tips to their original locations in their tip rack.

The default is `True`, so you only have to set `trash` when you want the tip-returning behavior:

```
pipette.transfer(
    volume=100,
    source=plate["A1"],
    dest=plate["B1"],
    trash=False,
)
```

New in version 2.0.

Sign Up For Our Newsletter

© OPENTRONS 2023

Labware and Deck Positions

The API automatically determines how the robot needs to move when working with the instruments and labware in your protocol. But sometimes you need direct control over these activities. The API lets you do just that. Specifically, you can control movements relative to labware and deck locations. You can also manage the gantry's speed and trajectory as it traverses the working area. This document explains how to use API commands to take direct control of the robot and position it exactly where you need it.

Position Relative to Labware

When the robot positions itself relative to a piece of labware, where it moves is determined by the labware definition, the actions you want it to perform, and the labware offsets for a specific deck slot. This section describes how these positional components are calculated and how to change them.

Top, Bottom, and Center

Every well on every piece of labware has three addressable positions: top, bottom, and center. The position is determined by the labware definition and what the labware is loaded on top of. You can use these positions as-is or calculate other positions relative to them.

Top

Let's look at the `Well.top()` method. It returns a position level with the top of the well, centered in both horizontal directions.

```
plate['A1'].top()  # the top center of the well
```

This is a good position to use for a blow out operation or an activity where you don't want the tip to contact the liquid. In addition, you can adjust the height of this position with the optional argument `z`, which is measured in mm. Positive `z` numbers move the position up, negative `z` numbers move it down.

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

New in version 2.0.

Bottom

Let's look at the `Well.bottom()` method. It returns a position level with the bottom of the well, centered in both horizontal directions.

```
plate['A1'].bottom()  # the bottom center of the well
```

This is a good position for aspirating liquid or an activity where you want the tip to cor

Hi there! What brings you to Opentrons today?

`Well.top()` method, you can adjust the height of this position with the optional argument `z`, which is measured in mm. Positive `z` numbers move the position up, negative `z` numbers move it down.

Warning:

Negative `z` arguments to `Well.bottom()` will cause the pipette tip to collide with the bottom of the well. Collisions may bend the tip (affecting liquid handling) and the pipette may be higher than expected on the z-axis until it picks up another tip.

Flex can detect collisions, and even gentle contact may trigger an overpressure error and cause the protocol to fail. Avoid `z` values less than 1, if possible.

The OT-2 has no sensors to detect contact with a well bottom. The protocol will continue even after a collision.

New in version 2.0.

Center

Let's look at the **`Well.center()`** method. It 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() # the vertical and horizontal center of the well
```

New in version 2.0.

Default Positions

By default, your robot will aspirate and dispense 1 mm above the bottom of wells. This default clearance may not be suitable for some labware geometries, liquids, or protocols. You can change this value by using the **`Well.bottom()`** method with the `z` argument, though it can be cumbersome to do so repeatedly.

If you need to change the aspiration or dispensing height for multiple operations, specify the distance in mm from the well bottom with the **`InstrumentContext.well_bottom_clearance`** object. It has two attributes: `well_bottom_clearance.aspirate` and `well_bottom_clearance.dispense`. These change the aspiration height and dispense height, respectively.

Modifying these attributes will affect all subsequent aspirate and dispense actions performed by the attached pipette, even those executed as part of a **`transfer()`** operation. This snippet from a sample protocol demonstrates how to work with and change the default clearance:

```
# aspirate 1 mm above the bottom of the well (default)
pipette.aspirate(50, plate['A1'])
# dispense 1 mm above the bottom of the well (default)
pipette.dispense(50, plate['A1'])

# change clearance for aspiration to 2 mm
pipette.well_bottom_clearance.aspirate = 2
# aspirate 2 mm above the bottom of the well
pipette.aspirate(50, plate['A1'])
# still dispensing 1 mm above the bottom
pipette.dispense(50, plate['A1'])

pipette.aspirate(50, plate['A1'])
# change clearance for dispensing to 10 mm
pipette.well_bottom_clearance.dispense = 10
# dispense high above the well
pipette.dispense(50, plate['A1'])
```

New in version 2.0.

Using Labware Position Check

deck slot, even across different protocols.

You should only adjust labware offsets in your Python code if you plan to run your protocol in Jupyter Notebook or from the command line. See Setting Labware Offsets in the Advanced Control article for information.

Position Relative to the Deck

The robot's base coordinate system is known as *deck coordinates*. Many API functions use this coordinate system, and you can also reference it directly. It is a right-handed coordinate system always specified in mm, with the origin `(0, 0, 0)` at the front left of the robot. The positive `x` direction is to the right, the positive `y` direction is to the back, and the positive `z` direction is up.

You can identify a point in this coordinate system with a **`types.Location`** object, either as a standard Python **`tuple`** of three floats, or as an instance of the **`namedtuple`** **`types.Point`**.

> **Note:**
>
> There are technically multiple vertical axes. For example, `z` is the axis of the left pipette mount and `a` is the axis of the right pipette mount. There are also pipette plunger axes: `b` (left) and `c` (right). You usually don't have to refer to these axes directly, since most motion commands are issued to a particular pipette and the robot automatically selects the correct axis to move. Similarly, **`types.Location`** only deals with `x`, `y`, and `z` values.

Independent Movement

For convenience, many methods have location arguments and incorporate movement automatically. This section will focus on moving the pipette independently, without performing other actions like `aspirate()` or `dispense()`.

Move To

The **`InstrumentContext.move_to()`** method moves a pipette to any reachable location on the deck. If the pipette has picked up a tip, it will move the end of the tip to that position; if it hasn't, it will move the pipette nozzle to that position.

The **`move_to()`** method requires the **`Location`** argument. The location can be automatically generated by methods like `Well.top()` and `Well.bottom()` or one you've created yourself, but you can't move a pipette to a well directly:

```
pipette.move_to(plate['A1'])              # error; can't move to a well itself
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(z=2))  # move to 2 mm above the bottom of well A1
pipette.move_to(plate['A1'].top(z=-2))    # move to 2 mm below the top of well A1
```

When using `move_to()`, by default the pipette will move in an arc: first upwards, then laterally to a position above the target location, and finally downwards to the target location. If you have a reason for doing so, you can force the pipette to move in a straight line to the target location:

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

> **Warning:**
>
> Moving without an arc runs the risk of the pipette colliding with objects on the deck. Be very careful when using this option, especially when moving longer distances.

```
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.

Points and Locations

When instructing the robot to move, it's important to consider the difference between the **`Point`** and **`Location`** types.

- Points are ordered tuples or named tuples: `Point(10, 20, 30)`, `Point(x=10, y=20, z=30)`, and `Point(z=30, y=20, x=10)` are all equivalent.
- Locations are a higher-order tuple that combines a point with a reference object: a well, a piece of labware, or `None` (the deck).

This distinction is important for the **`Location.move()`** method, which operates on a location, takes a point as an argument, and outputs an updated location. To use this method, include `from opentrons import types` at the start of your protocol. The `move()` method does not mutate the location it is called on, so to perform an action at the updated location, use it as an argument of another method or save it to a variable. For example:

```
# get the location at 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))

# aspirate 1 mm right, 1 mm back, and 1 mm up from the center of well A1
pipette.aspirate(50, adjusted_location)

# dispense at the same location
pipette.dispense(50, center_location.move(types.Point(x=1, y=1, z=1)))
```

Note:

The additional `z` arguments of the `top()` and `bottom()` methods (see Position Relative to Labware above) are shorthand for adjusting the top and bottom locations with `move()`. You still need to use `move()` to adjust these positions along the x- or y-axis:

```
# the following are equivalent
pipette.move_to(plate['A1'].bottom(z=2))
pipette.move_to(plate['A1'].bottom().move(types.Point(z=2)))

# adjust along the y-axis
pipette.move_to(plate['A1'].bottom().move(types.Point(y=2)))
```

New in version 2.0.

Movement Speeds

In addition to instructing the robot where to move a pipette, you can also control the speed at which it moves. Speed controls can be applied either to all pipette motions or to movement along a particular axis.

Gantry Speed

The robot's gantry usually moves as fast as it can given its construction. The default speed for Flex varies between 300 and 350 mm/s. The OT-2 default is 400 mm/s. However, some experiments or liquids may require slower movements. In this

```
pipette.move_to(plate['D6'].top())  # move to the last well at the slower speed
```

Warning:

These default speeds were chosen because they're the maximum speeds that Opentrons knows will work with the gantry. Your robot may be able to move faster, but you shouldn't increase this value unless instructed by Opentrons Support.

New in version 2.0.

Axis Speed Limits

In addition to controlling the overall gantry speed, you can set speed limits for each of the individual axes: `x` (gantry left/right motion), `y` (gantry forward/back motion), `z` (left pipette up/down motion), and `a` (right pipette up/down motion). Unlike `default_speed`, which is a pipette property, axis speed limits are stored in a protocol property `ProtocolContext.max_speeds`; therefore the `x` and `y` values affect all movements by both pipettes. This property works like a dictionary, where the keys are axes, assigning a value to a key sets a max speed, and deleting a key or setting it to `None` resets that axis's limit to the default:

```
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
```

Note that `max_speeds` can't set limits for the pipette plunger axes (`b` and `c`); instead, set the flow rates or plunger speeds as described in Pipette Flow Rates.

New in version 2.0.

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Advanced Control

As its name implies, the Python Protocol API is primarily designed for creating protocols that you upload via the Opentrons App and execute on the robot as a unit. But sometimes it's more convenient to control the robot outside of the app. For example, you might want to have variables in your code that change based on user input or the contents of a CSV file. Or you might want to only execute part of your protocol at a time, especially when developing or debugging a new protocol.

The Python API offers two ways of issuing commands to the robot outside of the app: through Jupyter Notebook or on the command line with `opentrons_execute`.

Jupyter Notebook

The Flex and OT-2 run Jupyter Notebook servers on port 48888, which you can connect to with your web browser. This is a convenient environment for writing and debugging protocols, since you can define different parts of your protocol in different notebook cells and run a single cell at a time.

Access your robot's Jupyter Notebook by either:

- Going to the **Advanced** tab of Robot Settings and clicking **Launch Jupyter Notebook**.
- Going directly to `http://<robot-ip>:48888` in your web browser (if you know your robot's IP address).

Once you've launched Jupyter Notebook, you can create a notebook file or edit an existing one. These notebook files are stored on the the robot. If you want to save code from a notebook to your computer, go to **File > Download As** in the notebook interface.

Protocol Structure

Jupyter Notebook is structured around *cells*: discrete chunks of code that can be run individually. This is nearly the opposite of Opentrons protocols, which bundle all commands into a single `run` function. Therefore, to take full advantage of Jupyter Notebook, you have to restructure your protocol.

Rather than writing a `run` function and embedding commands within it, start your notebook by importing `opentrons.execute` and calling **`opentrons.execute.get_protocol_api()`**. This function also replaces the `metadata` block of a standalone protocol by taking the minimum API version as its argument. Then you can call **`ProtocolContext`** methods in subsequent lines or cells:

```
import opentrons.execute
protocol = opentrons.execute.get_protocol_api('2.15')
protocol.home()
```

The first command you execute should always be **`home()`**. If you try to execute other commands first, you will get a `MustHomeError`. (When running protocols through the Opentrons App, the robot homes automatically.)

You should use the same **`ProtocolContext`** throughout your notebook, unless you need to start over from the beginning of your protocol logic. In that case, call **`get_protocol_api()`** again to get a new **`ProtocolContext`**.

Running a Previously Written Protocol

You can also use Jupyter to run a protocol that you have already written. To do so, first copy the entire text of the protocol into a cell and run that cell:

```
import opentrons.execute
from opentrons import protocol_api
def run(protocol: protocol_api.ProtocolContext):
    # the contents of your previously written protocol go here
```

Since a typical protocol only *defines* the `run` function but doesn't *call* it, this won't immediately cause the robot to move. To begin the run, instantiate a **`ProtocolContext`** and pass it to the `run` function you just defined:

```
protocol = opentrons.execute.get_protocol_api('2.15')
run(protocol)  # your protocol will now run
```

Setting Labware Offsets

All positions relative to labware are adjusted automatically based on labware offset data. When you're running your code in Jupyter Notebook or with `opentrons_execute`, you need to set your own offsets because you can't perform run setup and Labware Position Check in the Opentrons App or on the Flex touchscreen. For these applications, do the following to calculate and apply labware offsets:

1. Create a "dummy" protocol that loads your labware and has each used pipette pick up a tip from a tip rack.
2. Import the dummy protocol to the Opentrons App.
3. Run Labware Position Check from the app or touchscreen.
4. Add the offsets to your code with **`set_offset()`**.

Creating the dummy protocol requires you to:

1. Use the `metadata` or `requirements` dictionary to specify the API version. (See Versioning for details.) Use the same API version as you did in **`opentrons.execute.get_protocol_api()`**.
2. Define a `run()` function.
3. Load all of your labware in their initial locations.
4. Load your smallest capacity pipette and specify its `tip_racks`.
5. Call `pick_up_tip()`. Labware Position Check can't run if you don't pick up a tip.

```
metadata = {"apiLevel": "2.13"}

def run(protocol):
    tiprack = protocol.load_labware("opentrons_96_tiprack_300ul", 1)
    reservoir = protocol.load_labware("nest_12_reservoir_15ml", 2)
    plate = protocol.load_labware("nest_96_wellplate_200ul_flat", 3)
    p300 = protocol.load_instrument("p300_single_gen2", "left", tip_racks=[tiprack])
    p300.pick_up_tip()
    p300.return_tip()
```

After importing this protocol to the Opentrons App, run Labware Position Check to get the x, y, and z offsets for the tip rack and labware. When complete, you can click **Get Labware Offset Data** to view automatically generated code that uses `set_offset()` to apply the offsets to each piece of labware.

```
labware_1 = protocol.load_labware("opentrons_96_tiprack_300ul", location="1")
labware_1.set_offset(x=0.00, y=0.00, z=0.00)

labware_2 = protocol.load_labware("nest_12_reservoir_15ml", location="2")
labware_2.set_offset(x=0.10, y=0.20, z=0.30)

labware_3 = protocol.load_labware("nest_96_wellplate_200ul_flat", location="3")
labware_3.set_offset(x=0.10, y=0.20, z=0.30)
```

This automatically generated code uses generic names for the loaded labware. If you want to match the labware names already in your protocol, change the labware names to match your original code:

```
reservoir = protocol.load_labware("nest_12_reservoir_15ml", "2")
reservoir.set_offset(x=0.10, y=0.20, z=0.30)
```

New in version 2.12.

Once you've executed this code in Jupyter Notebook, all subsequent positional calculations for this reservoir in slot 2 will be adjusted 0.1 mm to the right, 0.2 mm to the back, and 0.3 mm up.

Remember, you should only add `set_offset()` commands to protocols run outside of the Opentrons App. And you should follow the behavior of Labware Position Check, i.e., *do not* reuse offset measurements unless they apply to the *same labware* in the *same deck slot* on the *same robot*.

> **Warning:**
>
> Improperly reusing offset data may cause your robot to move to an unexpected position or crash against labware, which can lead to incorrect protocol execution or damage your equipment. The same applies when running protocols with `set_offset()` commands in the Opentrons App. When in doubt: run Labware Position Check again and update your code!

Using 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()`.

Using Modules

If your protocol uses modules, you need to take additional steps to make sure that Jupyter Notebook doesn't send commands that conflict with the robot server. Sending commands to modules while the robot server is running will likely cause errors, and the module commands may not execute as expected.

To disable the robot server, open a Jupyter terminal session by going to **New > Terminal** and run `systemctl stop opentrons-robot-server`. Then you can run code from cells in your notebook as usual. When you are done using Jupyter Notebook, you should restart the robot server with `systemctl start opentrons-robot-server`.

> **Note:**
>
> While the robot server is stopped, the robot will display as unavailable in the Opentrons App. If you need to control the robot or its attached modules through the app, you need to restart the robot server and wait for the robot to appear as available in the app.

Command Line

The robot's command line is accessible either by going to **New > Terminal** in Jupyter or via SSH.

To execute a protocol from the robot's command line, copy the protocol file to the robot with `scp` and then run the protocol with `opentrons_execute`:

```
$ opentrons_execute /data/my_protocol.py
```

By default, `opentrons_execute` will print out the same run log shown in the Opentrons App, as the protocol executes. It also prints out internal logs at the level `warning` or above. Both of these behaviors can be changed. Run `opentrons_execute --help` for more information.

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

Protocol Examples

This page provides simple, ready-made protocols for Flex and OT-2. Feel free to copy and modify these examples to create unique protocols that help automate your laboratory workflows. Also, experimenting with these protocols is another way to build upon the skills you've learned from working through the tutorial. Try adding different hardware, labware, and commands to a sample protocol and test its validity after importing it into the Opentrons App.

Using These Protocols

These sample protocols are designed for anyone using an Opentrons Flex or OT-2 liquid handling robot. For our users with little to no Python experience, we've taken some liberties with the syntax and structure of the code to make it easier to understand. For example, we've formatted the samples with line breaks to show method arguments clearly and to avoid horizontal scrolling. Additionally, the methods use named arguments instead of positional arguments. For example:

```
# This code uses named arguments
tiprack_1 = protocol.load_labware(
    load_name='opentrons_flex_96_tiprack_200ul',
    location='D2')

# This code uses positional arguments
tiprack_1 = protocol.load_labware('opentrons_flex_96_tiprack_200ul','D2')
```

Both examples instantiate the variable `tiprack_1` with a Flex tip rack, but the former is more explicit. It shows the parameter name and its value together (e.g. `location='D2'`), which may be helpful when you're unsure about what's going on in a protocol code sample.

Python developers with more experience should feel free to ignore the code styling used here and work with these examples as you like.

Instruments and Labware

The sample protocols all use the following pipettes:

- Flex 1-Channel Pipette (5–1000 µL). The API load name for this pipette is `flex_1channel_1000`.
- P300 Single-Channel GEN2 pipette for the OT-2. The API load name for this pipette is `p300_single_gen2`.

They also use the labware listed below:

Labware type	Labware name	API load name
Reservoir	USA Scientific 12-Well Reservoir 22 mL	`usascientific_12_reservoir_22ml`
Well plate	Corning 96-Well Plate 360 µL Flat	`corning_96_wellplate_360ul_flat`
Flex tip rack	Opentrons Flex 96 Tip Rack 200 µL	`opentrons_flex_96_tiprack_200ul`
OT-2 tip rack	Opentrons 96 Tip Rack 300 µL	`opentrons_96_tiprack_300ul`

Protocol Template

Flex | OT-2

```python
from opentrons import protocol_api

requirements = {"robotType": "Flex", "apiLevel": "2.15"}

def run(protocol: protocol_api.ProtocolContext):
    # load tip rack in deck slot D3
    tiprack = protocol.load_labware(
        load_name="opentrons_flex_96_tiprack_1000ul", location="D3"
    )
    # attach pipette to left mount
    pipette = protocol.load_instrument(
        instrument_name="flex_1channel_1000",
        mount="left",
        tip_racks=[tiprack]
    )
    # load well plate in deck slot D2
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat", location="D2"
    )
    # load reservoir in deck slot D1
    reservoir = protocol.load_labware(
        load_name="usascientific_12_reservoir_22ml", location="D1"
    )
    # Put protocol commands here
```

Transferring Liquids

These protocols demonstrate how to move 100 µL of liquid from one well to another.

Basic Method

This protocol uses some building block commands to tell the robot, explicitly, where to go to aspirate and dispense liquid. These commands include the `pick_up_tip()`, `aspirate()`, and `dispense()` methods.

Flex | OT-2

```python
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel':'2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
    tiprack_1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D2')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
    tip_racks=[tiprack_1])

    pipette_1.pick_up_tip()
    pipette_1.aspirate(100, plate['A1'])
    pipette_1.dispense(100, plate['B1'])
    pipette_1.drop_tip()
```

the `InstrumentContext.transfer()` method to move liquid between well plates. The source and destination well arguments (e.g., `plate['A1'], plate['B1']`) are part of `transfer()` method parameters. You don't need separate calls to `aspirate` or `dispense` here.

Flex | OT-2

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel': '2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
    tiprack_1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D2')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack_1])
    # transfer 100 µL from well A1 to well B1
    pipette_1.transfer(100, plate['A1'], plate['B1'])
```

Loops

In Python, a loop is an instruction that keeps repeating an action until a specific condition is met.

When used in a protocol, loops automate repetitive steps such as aspirating and dispensing liquids from a reservoir to a a range of wells, or all the wells, in a well plate. For example, this code sample loops through the numbers 0 to 7, and uses the loop's current value to transfer liquid from all the wells in a reservoir to all the wells in a 96-well plate.

Flex | OT-2

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel':'2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
    tiprack_1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D2')
    reservoir = protocol.load_labware(
        load_name='usascientific_12_reservoir_22ml',
        location='D3')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack_1])

    # distribute 20 µL from reservoir:A1 -> plate:row:1
    # distribute 20 µL 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):
        pipette_1.distribute(200, reservoir.wells()[i], plate.rows()[i])
```

Multiple Air Gaps

Opentrons electronic pipettes can do some things that a human cannot do with a pipette, like accurately alternate between liquid and air aspirations that create gaps within the same tip. The protocol shown below shows you how to aspirate from the first five wells in the reservoir and create an air gap between each sample.

Flex | OT-2

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel':'2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
    tiprack_1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D2')
    reservoir = protocol.load_labware(
        load_name='usascientific_12_reservoir_22ml',
        location='D3')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack_1])

    pipette_1.pick_up_tip()

    # aspirate from the first 5 wells
    for well in reservoir.wells()[:4]:
        pipette_1.aspirate(volume=35, location=well)
        pipette_1.air_gap(10)

    pipette_1.dispense(225, plate['A1'])

    pipette_1.return_tip()
```

Notice here how Python's `slice` functionality (in the code sample as `[:4]`) lets us select the first five wells of the well plate only. Also, in Python, a range of numbers is *exclusive* of the end value and counting starts at 0, not 1. For the Corning 96-well plate used here, this means well A1=0, B1=1, C1=2, and so on to the last well used, which is E1=4. See also, the Commands section of the Tutorial.

Dilution

This protocol dispenses diluent to all wells of a Corning 96-well plate. Next, it dilutes 8 samples from the reservoir across all 8 columns of the plate.

Flex | OT-2

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel': '2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
```

```
        location='D3')
    reservoir = protocol.load_labware(
        load_name='usascientific_12_reservoir_22ml',
        location='C1')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='left',
        tip_racks=[tiprack_1, tiprack_2])
    # Dispense diluent
    pipette_1.distribute(50, reservoir['A12'], plate.wells())

    # 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 30 µL of source to first well in column
    pipette_1.transfer(30, source, row[0], mix_after=(3, 25))

    # dilute the sample down the column
    pipette_1.transfer(
        30, row[:11], row[1:],
        mix_after=(3, 25))
```

Notice here how the code sample loops through the rows and uses slicing to distribute the diluent. For information about these features, see the Loops and Air Gaps examples above. See also, the Commands section of the Tutorial.

Plate Mapping

This protocol dispenses different volumes of liquids to a well plate and automatically refills the pipette when empty.

Flex | OT-2

```
from opentrons import protocol_api

requirements = {'robotType': 'Flex', 'apiLevel': '2.15'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name='corning_96_wellplate_360ul_flat',
        location='D1')
    tiprack_1 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D2')
    tiprack_2 = protocol.load_labware(
        load_name='opentrons_flex_96_tiprack_200ul',
        location='D3')
    reservoir = protocol.load_labware(
        load_name='usascientific_12_reservoir_22ml',
        location='C1')
    pipette_1 = protocol.load_instrument(
        instrument_name='flex_1channel_1000',
        mount='right',
    tip_racks=[tiprack_1, tiprack_2])

    # Volume amounts are for demonstration purposes only
    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,
```

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023

API Version 2 Reference

Protocols and Instruments

class opentrons.protocol_api.**ProtocolContext**(*api_version: APIVersion, core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore], broker: Optional[LegacyBroker] = None, core_map: Optional[LoadedCoreMap] = None, deck: Optional[Deck] = None, bundled_data: Optional[Dict[str, bytes]] = 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**: *APIVersion*

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.

property **bundled_data**: *Dict[str, bytes]*

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.

commands(*self*) → 'List[str]'

Return the run log.

This is a list of human-readable strings representing what's been done in the protocol so far. For example, "Aspirating 123 µL from well A1 of 96 well plate in slot 1."

The exact format of these entries is not guaranteed. The format here may differ from other places that show the run log, such as the Opentrons App.

New in version 2.0.

comment(*self, msg: 'str'*) → 'None'

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.

property **deck**: *Deck*

An interface to provide information about what's currently loaded on the deck. This object is useful for determining if a slot in the deck is free.

The value will be a `Labware` if the slot contains a labware, a `ModuleContext` if the slot contains a hardware module, or `None` if the slot doesn't contain anything.

Rather than filtering the objects in the deck map yourself, you can also use `loaded_labwares` to get a dict of labwares and `loaded_modules` to get a dict of modules.

For Advanced Control *only*, you can delete an element of the `deck` dict. This only works for deck slots that contain labware objects. For example, if slot 1 contains a labware, `del protocol.deck['1']` will free the slot so you can load another labware there.

> **Warning:**
>
> Deleting labware from a deck slot does not pause the protocol. Subsequent commands continue immediately. If you need to physically move the labware to reflect the new deck state, add a `pause()` or use `move_labware()` instead.

Changed in version 2.15: `del` sets the corresponding labware's location to `OFF_DECK`.

New in version 2.0.

define_liquid(*self*, *name: 'str'*, *description: 'Optional[str]'*, *display_color: 'Optional[str]'*) → 'Liquid'

Define a liquid within a protocol.

Parameters::
- **name** (*str*) – A human-readable name for the liquid.
- **description** (*str*) – An optional description of the liquid.
- **display_color** (*str*) – An optional hex color code, with hash included, to represent the specified liquid. Standard three-value, four-value, six-value, and eight-value syntax are all acceptable.

Returns:: A `Liquid` object representing the specified liquid.

New in version 2.14.

delay(*self*, *seconds: 'float' = 0*, *minutes: 'float' = 0*, *msg: 'Optional[str]' = None*) → '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.

property **door_closed**: *bool*

Returns True if the robot door is closed

New in version 2.5.

property **fixed_trash**: *Labware*

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*) → 'None'

Homes the robot.

New in version 2.0.

is_simulating(*self*) → 'bool'

New in version 2.0.

load_adapter(*self*, *load_name: 'str'*, *location: 'Union[DeckLocation, OffDeckType]'*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

ization (creating the adapter and adding it to the protocol) into one.

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

Parameters::
- **load_name** (*str*) – A string to use for looking up a labware definition for the adapter. You can find the `load_name` for any standard adapter on the Opentrons Labware Library.
- **location** (int or str or `OFF_DECK`) – Either a deck slot, like `1`, `"1"`, or `"D1"`, or the special value `OFF_DECK`.
- **namespace** (*str*) –
  The namespace that the labware definition belongs to. If unspecified, will search both:
  - `"opentrons"`, to load standard Opentrons labware definitions.
  - `"custom_beta"`, to load custom labware definitions created with the Custom Labware Creator.

  You might need to specify an explicit `namespace` if you have a custom definition whose `load_name` is the same as an Opentrons standard definition, and you want to explicitly choose one or the other.
- **version** – The version of the labware definition. You should normally leave this unspecified to let the implementation choose a good default.

New in version 2.15.

load_adapter_from_definition(*self*, *adapter_def: "'LabwareDefinition'"*, *location: 'Union[DeckLocation, OffDeckType]'*) → *'Labware'*

Specify the presence of an adapter on the deck.

This function loads the adapter definition specified by `adapter_def` to the location specified by `location`.

Parameters::
- **adapter_def** – The adapter's labware definition.
- **location** (int or str or `OFF_DECK`) – The slot into which to load the labware, such as `1`, `"1"`, or `"D1"`. See Deck Slots.

New in version 2.15.

load_instrument(*self*, *instrument_name: 'str'*, *mount: 'Union[Mount, str]'*, *tip_racks: 'Optional[List[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* or *str*) – The mount in which this instrument should be attached. This can either be an instance of the enum type `types.Mount` or one of the strings *'left'* and *'right'*.
- **tip_racks** (List[`Labware`]) – A list of tip racks from which to pick tips if `InstrumentContext.pick_up_tip()` 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[DeckLocation, OffDeckType]'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → *'Labware'*

Load a labware onto a location.

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** (*str*) –
  A string to use for looking up a labware definition. You can find the `load_name` for any standard labware on the Opentrons Labware Library.

- **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 that the labware definition belongs to. If unspecified, will search both:
  - `"opentrons"`, to load standard Opentrons labware definitions.
  - `"custom_beta"`, to load custom labware definitions created with the Custom Labware Creator.

  You might need to specify an explicit `namespace` if you have a custom definition whose `load_name` is the same as an Opentrons standard definition, and you want to explicitly choose one or the other.
- **version** – The version of the labware definition. You should normally leave this unspecified to let the implementation choose a good default.
- **adapter** – Load name of an adapter to load the labware on top of. The adapter will be loaded from the same given namespace, but version will be automatically chosen.

New in version 2.0.

load_labware_by_name(*self*, *load_name: 'str'*, *location: 'DeckLocation'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'int' = 1*) → *'Labware'*

Deprecated since version 2.0: Use **load_labware()** instead.

New in version 2.0.

load_labware_from_definition(*self*, *labware_def: "'LabwareDefinition'"*, *location: 'Union[DeckLocation, OffDeckType]'*, *label: 'Optional[str]' = None*) → *'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 or **OFF_DECK**) – The slot into which to load the labware, such as `1`, `"1"`, or `"D1"`. See Deck Slots.
- **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: 'Optional[DeckLocation]' = None*, *configuration: 'Optional[str]' = None*) → *'ModuleTypes'*

Load a module onto the deck, given its name or model.

This is the function to call to use a module in your protocol, like **load_instrument()** 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 by using **loaded_modules**.

Parameters::
- **module_name** (*str*) – The name or model of the module. See Available Modules for possible values.
- **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, like `1`, `"1"`, or `"D1"`. See Deck Slots.
  The Thermocycler is only valid in one deck location. You don't have to specify a location when loading it, but if you do, it must be `7`, `"7"`, or `"B1"`. See Thermocycler Module.
  Changed in version 2.15: You can now specify a deck slot as a coordinate, like `"D1"`.
- **configuration** –
  Configure a thermocycler to be in the `semi` position. This parameter does not work. Do not use it.

`MagneticModuleContext`, `TemperatureModuleContext`, or `ThermocyclerContext`, depending on what you requested with `module_name`.

Changed in version 2.13: Added `HeaterShakerContext` return value.

Changed in version 2.15: Added `MagneticBlockContext` return value.

New in version 2.0.

property **loaded_instruments**: *Dict[str, InstrumentContext]*

Get the instruments that have been loaded into the protocol.

This is a map of mount name to instruments previously loaded with `load_instrument()`. 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()`, the unused one will not be present.

Returns:: A dict mapping mount name (`'left'` or `'right'`) to the instrument in that mount. If a mount has no loaded instrument, that key will be missing from the dict.

New in version 2.0.

property **loaded_labwares**: *Dict[int, Labware]*

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 `module.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`.

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

New in version 2.0.

property **loaded_modules**: *Dict[int, Union[TemperatureModuleContext, MagneticModuleContext, ThermocyclerContext, HeaterShakerContext, MagneticBlockContext]]*

Get the modules loaded into the protocol context.

This is a map of deck positions to modules loaded by previous calls to `load_module()`. 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()`, only the Temperature Module will be present.

Returns Dict[int, 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**: *AxisMaxSpeeds*

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
```

> **Caution:**
>
> This property is not yet supported on API version 2.14 or higher.

New in version 2.0.

move_labware(*self*, *labware: 'Labware'*, *new_location: 'Union[DeckLocation, Labware, ModuleTypes, OffDeckType, WasteChute]'*, *use_gripper: 'bool' = False*, *pick_up_offset: 'Optional[Mapping[str, float]]' = None*, *drop_offset: 'Optional[Mapping[str, float]]' = None*) → 'None'

Move a loaded labware to a new location. See Moving Labware for more details.

Parameters::
- **labware** – The labware to move. It should be a labware already loaded using `load_labware()`.
- **new_location** –
  Where to move the labware to. This is either:
  - A deck slot like `1`, `"1"`, or `"D1"`. See Deck Slots.
  - A hardware module that's already been loaded on the deck with `load_module()`.
  - A labware or adapter that's already been loaded on the deck with `load_labware()` or `load_adapter()`.
  - The special constant `OFF_DECK`.
- **use_gripper** –
  Whether to use the Flex Gripper for this movement.
  - If `True`, will use the gripper to perform an automatic movement. This will raise an error on an OT-2 protocol.
  - If `False`, will pause protocol execution until the user performs the movement. Protocol execution remains paused until the user presses **Confirm and resume**.

Gripper-only parameters:

Parameters::
- **pick_up_offset** – Optional x, y, z vector offset to use when picking up labware.
- **drop_offset** – Optional x, y, z vector offset to use when dropping off labware.

Before moving a labware to or from a hardware module, make sure that the labware's current and new locations are accessible, i.e., open the Thermocycler lid or open the Heater-Shaker's labware latch.

New in version 2.15.

pause(*self*, *msg: 'Optional[str]' = None*) → 'None'

Pause execution of the protocol until it's resumed.

A human can resume the protocol through the Opentrons App.

This function returns immediately, but the next function call that is blocked by a paused robot (anything that involves moving) will not return until the protocol is resumed.

Parameters:: **msg** (*str*) – An optional message to show to connected clients. The Opentrons App will show this in the run log.

New in version 2.0.

property **rail_lights_on**: *bool*

Returns True if the rail lights are on

New in version 2.5.

resume(*self*) → 'None'

Resume the protocol after `pause()`.

Deprecated since version 2.12: The Python Protocol API supports no safe way for a protocol to resume itself. See https://github.com/Opentrons/opentrons/issues/8209. If you're looking for a way for your protocol to resume automatically after a period of time, use `delay()`.

New in version 2.0.

New in version 2.5.

class opentrons.protocol_api.**InstrumentContext**(*core: AbstractInstrument[AbstractWellCore], protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore], broker: LegacyBroker, api_version: APIVersion, tip_racks: List[Labware], trash: Optional[Labware], requested_as: str*)

A context for a specific pipette or instrument.

The InstrumentContext class provides the objects, attributes, and methods that allow you to use pipettes in your protocols.

Methods generally fall into one of two categories.

- They can change the state of the InstrumentContext object, like how fast it moves liquid or where it disposes of used tips.
- They can command the instrument to perform an action, like picking up tips, moving to certain locations, and aspirating or dispensing liquid.

Objects in this class should not be instantiated directly. Instead, instances are returned by `ProtocolContext.load_instrument()`.

New in version 2.0.

air_gap(*self, volume: 'Optional[float]' = None, height: 'Optional[float]' = None*) → 'InstrumentContext'

Draw air into the pipette's tip at the current well.

See Air Gap.

Parameters::
- **volume** (*float*) – The amount of air, measured in µL. Calling `air_gap()` with no arguments uses the entire remaining volume in the pipette.
- **height** (*float*) – The height, in mm, to move above the current well before creating the air gap. The default is 5 mm above the current well.

Raises:: `UnexpectedTipRemovalError` – If no tip is attached to the pipette.

Raises:: **RuntimeError** – If location cache is `None`. This should happen if `air_gap()` is called without first calling a method that takes a location (e.g., **`aspirate()`**, **`dispense()`**)

Returns:: This instance.

> **Note:**
>
> Both `volume` and `height` are optional, but 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 a single, unnamed argument, it will always be interpreted as a volume.

New in version 2.0.

property **api_version**: *APIVersion*

New in version 2.0.

aspirate(*self, volume: 'Optional[float]' = None, location: 'Optional[Union[types.Location, labware.Well]]' = None, rate: 'float' = 1.0*) → 'InstrumentContext'

Draw liquid into a pipette tip.

See Aspirate for more details and examples.

Parameters::
- **volume** (*int* or *float*) – The volume to aspirate, measured in µL. If 0 or unspecified, defaults to the maximum volume for the pipette and its currently attached tip.
- **location** –
  Tells the robot where to aspirate from. The location can be a **`Well`** or a **`Location`**.
  - If the location is a **`Well`**, the robot will aspirate at or above the bottom center of the well. The distance (in mm) from the well bottom is specified by **`well_bottom_clearance.aspirate`**.

- **rate** (*float*) – A multiplier for the default flow rate of the pipette. Calculated as `rate` multiplied by `flow_rate.aspirate`. If not specified, defaults to 1.0. See Pipette Flow Rates.

Returns:: This instance.

> **Note:**
>
> If `aspirate` is called with a single, unnamed argument, it will treat that argument as `volume`. If you want to call `aspirate` with only `location`, specify it as a keyword argument:
>
> `pipette.aspirate(location=plate['A1'])`

New in version 2.0.

blow_out(*self, location: 'Optional[Union[types.Location, labware.Well]]' = None*) → 'InstrumentContext'

Blow an extra amount of air through a pipette's tip to clear it.

If `dispense()` is used to empty a pipette, usually a small amount of liquid remains in the tip. During a blowout, the pipette moves the plunger beyond its normal limits to help remove all liquid from the pipette tip. See Blow Out.

Parameters:: **location** (`Well` or `Location` or `None`) – The blowout location. If no location is specified, the pipette will blow out from its current position.

Raises:: **RuntimeError** – If no location is specified and the location cache is `None`. This should happen if `blow_out()` is called without first calling a method that takes a location, like `aspirate()` or `dispense()`.

Returns:: This instance.

New in version 2.0.

property **channels**: *int*

The number of channels on the pipette.

Possible values are 1, 8, or 96.

New in version 2.0.

configure_for_volume(*self, volume: 'float'*) → 'None'

Configure a pipette to handle a specific volume of liquid, measured in µL. The pipette enters a volume mode depending on the volume provided. Changing pipette modes alters properties of the instance of `InstrumentContext`, such as default flow rate, minimum volume, and maximum volume. The pipette remains in the mode set by this function until it is called again.

The Flex 1-Channel 50 µL and Flex 8-Channel 50 µL pipettes must operate in a low-volume mode to accurately dispense very small volumes of liquid. Low-volume mode can only be set by calling `configure_for_volume()`. See Volume Modes.

> **Note:**
>
> Changing a pipette's mode will reset its flow rates.

This function will raise an error if called when the pipette's tip contains liquid. It won't raise an error if a tip is not attached, but changing modes may affect which tips the pipette can subsequently pick up without raising an error.

This function will also raise an error if `volume` is outside of the minimum and maximum capacities of the pipette (e.g., setting `volume=1` for a Flex 1000 µL pipette).

Parameters:: **volume** (*float*) – The volume, in µL, that the pipette will prepare to handle.

New in version 2.15.

consolidate(*self, volume: 'Union[float, Sequence[float]]', source: 'List[labware.Well]', dest: 'labware.Well', *args: 'Any', **kwargs: 'Any'*) → 'InstrumentContext'

- **source** – A list of wells to aspirate liquid from.
- **dest** – A single well to dispense liquid into.
- **kwargs** – See `transfer()` and the Complex Liquid Handling Parameters page. Some parameters behave differently than when transferring. `disposal_volume` and `mix_before` are ignored.

Returns:: This instance.

New in version 2.0.

property **current_volume**: *float*

The current amount of liquid held in the pipette, measured in µL.

New in version 2.0.

property **default_speed**: *float*

The speed at which the robot's gantry moves in mm/s.

The default speed for Flex varies between 300 and 350 mm/s. The OT-2 default is 400 mm/s. In addition to changing the default, the speed of individual motions can be changed with the `speed` argument of the `InstrumentContext.move_to()` method. See Gantry Speed.

New in version 2.0.

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

Dispense liquid from a pipette tip.

See Dispense for more details and examples.

Parameters::

- **volume** (*int* or *float*) – The volume to dispense, measured in µL. If 0 or unspecified, defaults to `current_volume`. If only a volume is passed, the pipette will dispense from its current position.
- **location** –

  Tells the robot where to dispense liquid held in the pipette. The location can be a `Well` or a `Location`.
  - If the location is a `Well`, the pipette will dispense at or above the bottom center of the well. The distance (in mm) from the well bottom is specified by `well_bottom_clearance.dispense`.
  - If the location is a `Location` (e.g., the result of `Well.top()` or `Well.bottom()`), the robot will dispense into that specified position.
  - If the `location` is unspecified, the robot will dispense into its current position.

  If only a `location` is passed (e.g., `pipette.dispense(location=plate['A1'])`), all of the liquid aspirated into the pipette will be dispensed (the amount is accessible through `current_volume`).
- **rate** (*float*) – How quickly a pipette dispenses liquid. The speed in µL/s is calculated as `rate` multiplied by `flow_rate.dispense`. If not specified, defaults to 1.0. See Pipette Flow Rates.
- **push_out** (*float*) – Continue past the plunger bottom to help ensure all liquid leaves the tip. Measured in µL. The default value is `None`.

Returns:: This instance.

> **Note:**
>
> If `dispense` is called with a single, unnamed argument, it will treat that argument as `volume`. If you want to call `dispense` with only `location`, specify it as a keyword argument: `pipette.dispense(location=plate['A1'])`.

New in version 2.0.

distribute(*self, volume: 'Union[float, Sequence[float]]', source: 'labware.Well', dest: 'List[labware.Well]', *args: 'Any', **kwargs: 'Any'*) → *'InstrumentContext'*

Move a volume of liquid from one source to multiple destinations.

Parameters::

- **volume** – The amount, in µL, to dispense into each destination well.

See `transfer()` and the Complex Liquid Handling Parameters page. Some parameters behave differently than when transferring.

- `disposal_volume` aspirates additional liquid to improve the accuracy of each dispense. Defaults to the minimum volume of the pipette. See Disposal Volume for details.
- `mix_after` is ignored.

Returns:: This instance.

New in version 2.0.

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

Drop the current tip.

See Dropping a Tip for examples.

If no location is passed (e.g. `pipette.drop_tip()`), the pipette will drop the attached tip into its default `trash_container`.

Starting with API version 2.15, if the trash container is the default fixed trash, the API will instruct the pipette to drop tips in different locations within the trash container. Varying the tip drop location helps prevent tips from piling up in a single location.

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:

- As a `Well`. This uses a default location relative to the well. This style of call can be used to make the robot drop a tip into labware like a well plate or a reservoir. For example, `pipette.drop_tip(location=reservoir["A1"])`.
- As a `Location`. For example, to drop a tip from an unusually large height above the tip rack, you could call `pipette.drop_tip(tip_rack["A1"].top(z=10))`.

Parameters::
- **location** (`Location` or `Well` or `None`) – The location to drop the tip.
- **home_after** –
  Whether to home the pipette's plunger after dropping the tip. If not specified, defaults to `True` on an OT-2.
  When `False`, the pipette does not home its plunger. This can save a few seconds, but is not recommended. Homing helps the robot track the pipette's position.

Returns:: This instance.

New in version 2.0.

property **flow_rate**: *FlowRates*

The speeds, in µL/s, configured for the pipette.

See Pipette Flow Rates.

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

> **Note:**
>
> Setting values of `speed`, which is deprecated, will override the values in `flow_rate`.

New in version 2.0.

property **has_tip**: *bool*

Whether this instrument has a tip attached or not.

The value of this property is determined logically by the API, not by detecting the physical presence of a tip. This is the case even on Flex, which has sensors to detect tip attachment.

New in version 2.7.

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**: *PipetteDict*

View the information returned by the hardware API directly.

Raises:: types.PipetteNotAttachedError if the pipette is no longer attached (should not happen).

New in version 2.0.

property **max_volume**: *float*

The maximum volume, in µL, that the pipette can hold.

The maximum volume that you can actually aspirate might be lower than this, depending on what kind of tip is attached to this pipette. For example, a P300 Single-Channel pipette always has a max_volume of 300 µL, but if it's using a 200 µL filter tip, its usable volume would be limited to 200 µL.

New in version 2.0.

property **min_volume**: *float*

The minimum volume, in µL, that the pipette can hold. This value may change based on the volume mode that the pipette is currently configured for.

New in version 2.0.

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

Mix a volume of liquid by repeatedly aspirating and dispensing it in a single location.

See Mix for examples.

Parameters::
- **repetitions** – Number of times to mix (default is 1).
- **volume** – The volume to mix, measured in µL. If 0 or unspecified, defaults to the maximum volume for the pipette and its attached tip.
- **location** – The Well or Location where the pipette will mix. If unspecified, the pipette will mix at its current position.
- **rate** – How quickly the pipette aspirates and dispenses liquid while mixing. The aspiration flow rate is calculated as rate multiplied by flow_rate.aspirate. The dispensing flow rate is calculated as rate multiplied by flow_rate.dispense. See Pipette Flow Rates.

Raises:: UnexpectedTipRemovalError – If no tip is attached to the pipette.

Returns:: This instance.

> **Note:**
>
> All the arguments of mix are optional. However, if you omit one of them, all subsequent arguments must be passed as keyword arguments. For instance, pipette.mix(1, location=wellplate['A1']) is a valid call, but pipette.mix(1, wellplate['A1']) is not.

New in version 2.0.

property **model**: *str*

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

New in version 2.0.

New in version 2.0.

move_to(*self*, *location: 'types.Location'*, *force_direct: 'bool' = False*, *minimum_z_height: 'Optional[float]' = None*, *speed: 'Optional[float]' = None*, *publish: 'bool' = True*) → 'InstrumentContext'

Move the instrument.

See Move To for examples.

Parameters::
- **location** (`Location`) – The location to move to.
- **force_direct** –
  If `True`, move directly to the destination without arc motion.

  > **Warning:**
  > Forcing direct motion can cause the pipette to crash into labware, modules, or other objects on the deck.

- **minimum_z_height** – An amount, measured in mm, to raise the mid-arc height. The mid-arc height can't be lowered.
- **speed** – The speed at which to move. By default, `InstrumentContext.default_speed`. This controls the straight linear speed of the motion. To limit individual axis speeds, use `ProtocolContext.max_speeds`.
- **publish** – Whether to list this function call in the run preview. Default is `True`.

New in version 2.0.

property **name**: *str*

The name string for the pipette (e.g., `"p300_single"`).

New in version 2.0.

pick_up_tip(*self*, *location: 'Union[types.Location, labware.Well, labware.Labware, None]' = None*, *presses: 'Optional[int]' = None*, *increment: 'Optional[float]' = None*, *prep_after: 'Optional[bool]' = None*) → 'InstrumentContext'

Pick up a tip for the pipette to run liquid-handling commands.

See Picking Up a Tip.

If no location is passed, the pipette will pick up the next available tip in its `tip_racks` list. Within each tip rack, tips will be picked up in the order specified by the labware definition and `Labware.wells()`. To adjust where the sequence starts, use `starting_tip`.

Parameters::
- **location** (`Well` or `Labware` or `types.Location`) –
  The location from which to pick up a tip. The `location` argument can be specified in several ways:
  - As a `Well`. For example, `pipette.pick_up_tip(tiprack.wells()[0])` will always pick up the first tip in `tiprack`, even if the rack is not a member of `InstrumentContext.tip_racks`.
  - As a labware. `pipette.pick_up_tip(tiprack)` will pick up the next available tip in `tiprack`, even if the rack is not a member of `InstrumentContext.tip_racks`.
  - As a `Location`. Use this to make fine adjustments to the pickup location. For example, to tell the robot to start its pick up tip routine 1 mm closer to the top of the well in the tip rack, call `pipette.pick_up_tip(tiprack["A1"].top(z=-1))`.
- **presses** (*int*) –
  The number of times to lower and then raise the pipette when picking up a tip, to ensure a good seal. Zero (`0`) will result in the pipette hovering over the tip but not picking it up (generally not desirable, but could be used for a dry run).
  Deprecated since version 2.14: Use the Opentrons App to change pipette pick-up settings.
- **increment** (*float*) –

Deprecated since version 2.14: Use the Opentrons App to change pipette pick-up settings.

- **prep_after** (*bool*) –
  Whether the pipette plunger should prepare itself to aspirate immediately after picking up a tip.
  If `True`, the pipette will move its plunger position to bottom in preparation for any following calls to **`aspirate()`**.
  If `False`, the pipette will prepare its plunger later, during the next call to **`aspirate()`**. This is accomplished by moving the tip to the top of the well, and positioning the plunger outside any potential liquids.

  Warning:
  This is provided for compatibility with older Python Protocol API behavior. You should normally leave this unset.
  Setting `prep_after=False` may create an unintended pipette movement, when the pipette automatically moves the tip to the top of the well to prepare the plunger.

Changed in version 2.13: Adds the `prep_after` argument. In version 2.12 and earlier, the plunger can't prepare itself for aspiration during **`pick_up_tip()`**, and will instead always prepare during **`aspirate()`**. Version 2.12 and earlier will raise an `APIVersionError` if a value is set for `prep_after`.

Returns:: This instance.

New in version 2.0.

`reset_tipracks`(*self*) → `'None'`

Reload all tips in each tip rack and reset the starting tip.

New in version 2.0.

property **`return_height`**: *float*

The height to return a tip to its tip rack.

Returns:: A scaling factor to apply to the tip length. During **`drop_tip()`**, this factor is multiplied by the tip length to get the distance from the top of the well to drop the tip.

New in version 2.2.

`return_tip`(*self*, *home_after: 'Optional[bool]' = None*) → `'InstrumentContext'`

Drop the currently attached tip in its original location in the tip rack.

Returning a tip does not reset tip tracking, so **`Well.has_tip`** will remain `False` for the destination.

Returns:: This instance.

Parameters:: **home_after** – See the `home_after` parameter of **`drop_tip()`**.

New in version 2.0.

property **`speed`**: *PlungerSpeeds*

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:

Setting values of **`flow_rate`** will override the values in **`speed`**.

Changed in version 2.14: This property has been removed because it's fundamentally misaligned with the stepwise nature of a pipette's plunger speed configuration. Use **`flow_rate`** instead.

New in version 2.0.

Note:

In robot software versions 6.3.0 and 6.3.1, protocols specifying API level 2.14 ignored `starting_tip` on the second and subsequent calls to `InstrumentContext.pick_up_tip()` with no argument. This is fixed for all API levels as of robot software version 7.0.0.

New in version 2.0.

property **tip_racks**: *List[Labware]*

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()` without arguments. See Picking Up a Tip.

New in version 2.0.

touch_tip(*self, location: 'Optional[labware.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 leftover droplets.

See Touch Tip for more details and examples.

Parameters::
- **location** (`Well` or `None`) – If no location is passed, the pipette will touch its tip at the edges of the current well.
- **radius** (*float*) – How far to move, as a proportion of the target well's radius. When `radius=1.0`, the pipette tip will move all the way to the edge of the target well. When `radius=0.5`, it will move to 50% of the well's radius. Default is 1.0 (100%)
- **v_offset** (*float*) – How far above or below the well to touch the tip, measured in mm. A positive offset moves the tip higher above the well. A negative offset moves the tip lower into the well. Default is -1.0 mm.
- **speed** (*float*) –
  The speed for touch tip motion, in mm/s.
  - Default: 60.0 mm/s
  - Maximum: 80.0 mm/s
  - Minimum: 1.0 mm/s

Raises:: `UnexpectedTipRemovalError` – If no tip is attached to the pipette.

Raises:: **RuntimeError** – If no location is specified and the location cache is `None`. This should happen if `touch_tip` is called without first calling a method that takes a location, like `aspirate()` or `dispense()`.

Returns:: This instance.

New in version 2.0.

transfer(*self, volume: 'Union[float, Sequence[float]]', source: 'AdvancedLiquidHandling', dest: 'AdvancedLiquidHandling', trash: 'bool' = True, **kwargs: 'Any'*) → 'InstrumentContext'

Move liquid from one well or group of wells to another.

Transfer is a higher-level command, incorporating other `InstrumentContext` commands, like `aspirate()` and `dispense()`. It makes writing a protocol easier at the cost of specificity. See Complex Commands for details on how transfer and other complex commands perform their component steps.

Parameters::
- **volume** – The amount, in µL, to aspirate from each source and dispense to each destination. If `volume` is a list, each amount will be used for the source and destination at the matching index. A list item of `0` will skip the corresponding wells entirely. See List of Volumes for details and examples.
- **source** – A single well or a list of wells to aspirate liquid from.
- **dest** – A single well or a list of wells to dispense liquid into.

Keyword Arguments:: Transfer accepts a number of optional parameters that give you greater control over the exact steps it performs. See Complex Liquid Handling Parameters or the links un-

- `"once"`: Use one tip for the entire command.
- `"always"`: Use a new tip for each set of aspirate and dispense steps.
- `"never"`: Do not pick up or drop tips at all.

See Tip Handling for details.

- **trash** (*boolean*) – If `True` (default), the pipette will drop tips in its `trash_container()`. If `False`, the pipette will return tips to their tip rack.

  See Trash Tips for details.
- **touch_tip** (*boolean*) – If `True`, perform a `touch_tip()` following each `aspirate()` and `dispense()`. Defaults to `False`.

  See Touch Tip for details.
- **blow_out** (*boolean*) – If `True`, a `blow_out()` will occur following each `dispense()`, but only if the pipette has no liquid left in it. If `False` (default), the pipette will not blow out liquid.

  See Blow Out for details.
- **blowout_location** (*string*) – Accepts one of three string values: `"trash"`, `"source well"`, or `"destination well"`.

  If `blow_out` is `False` (its default), this parameter is ignored.

  If `blow_out` is `True` and this parameter is not set:

  - Blow out into the trash, if the pipette is empty or only contains the disposal volume.
  - Blow out into the source well, if the pipette otherwise contains liquid.
- **mix_before** (*tuple*) – Perform a `mix()` before each `aspirate()` during the transfer. The first value of the tuple is the number of repetitions, and the second value is the amount of liquid to mix in µL.

  See Mix Before for details.
- **mix_after** (*tuple*) – Perform a `mix()` after each `dispense()` during the transfer. The first value of the tuple is the number of repetitions, and the second value is the amount of liquid to mix in µL.

  See Mix After for details.
- **disposal_volume** (*float*) – Transfer ignores the numeric value of this parameter. If set, the pipette will not aspirate additional liquid, but it will perform a very small blow out after each dispense.

  See Disposal Volume for details.

Returns:: This instance.

New in version 2.0.

property **trash_container**: *Labware*

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()` or `blow_out()` without arguments.

By default, the trash container is in slot A3 on Flex and in slot 12 on OT-2.

New in version 2.0.

property **type**: *str*

One of `'single'` or `'multi'`.

New in version 2.0.

property **well_bottom_clearance**: *Clearances*

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

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

When `aspirate()` or `dispense()` is given a `Well` rather than a full `Location`, the robot will move this distance above the bottom of the well to aspirate or dispense.

New in version 2.0.

class opentrons.protocol_api.**Liquid**(*_id: str, name: str, description: Optional[str], display_color: Optional[str]*)

A liquid to load into a well.

name

A human-readable name for the liquid.

Type:: str

description

An optional description.

Type:: Optional[str]

display_color

An optional display color for the liquid.

Type:: Optional[str]

New in version 2.14.

Labware and Wells

class opentrons.protocol_api.**Labware**(*core: AbstractLabware[Any], api_version: APIVersion, protocol_core: ProtocolCore, core_map: LoadedCoreMap*)

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()`, `wells()`, `columns()`, `rows()`, `rows_by_name()`, and `columns_by_name()`. 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**: *APIVersion*

New in version 2.0.

property **calibrated_offset**: *Point*

New in version 2.0.

property **child**: *Optional[Labware]*

The labware (if any) present on this labware.

New in version 2.15.

columns(*self, *args: 'Union[int, str]'*) → 'List[List[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, 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

Deprecated since version 2.0: Use `columns_by_name()` instead.

New in version 2.0.

columns_by_name(*self*) → 'Dict[str, List[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, 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**: *float*

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_adapter**: *bool*

New in version 2.15.

property **is_tiprack**: *bool*

New in version 2.0.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Load a compatible labware onto the labware using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_labware` (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

New in version 2.15.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → 'Labware'

Load a labware onto 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 2.15.

property **load_name**: *str*

The API load name of the labware definition

New in version 2.0.

property **magdeck_engage_height**: *Optional[float]*

Return the default magnet engage height that `MagneticModuleContext.engage()` will use for this labware.

Warning:

This currently returns confusing and unpredictable results that do not necessarily match what `MagneticModuleContext.engage()` will actually choose for its default height.

The confusion is related to how this height's units and origin point are defined, and differences between Magnetic Module generations.

property **name**: *str*

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.

property **parameters**: *LabwareParameters*

Internal properties of a labware including type and quirks

New in version 2.0.

property **parent**: *Union[str, Labware, ModuleTypes, OffDeckType]*

The parent of this labware—where this labware is loaded.

Returns:: If the labware is directly on the robot's deck, the *str* name of the deck slot, like `"D1"` (Flex) or `"1"` (OT-2). See Deck Slots.
If the labware is on a module, a `ModuleContext`.
If the labware is on a labware or adapter, a `Labware`.
If the labware is off-deck, `OFF_DECK`.

Changed in version 2.14: Return type for module parent changed to `ModuleContext`. Prior to this version, an internal geometry interface is returned.

Changed in version 2.15: Will return a `Labware` if the labware was loaded onto a labware/adapter. Will now return `OFF_DECK` if the labware is off-deck. Formerly, if the labware was removed by using `del` on `deck`, this would return where it was before its removal.

New in version 2.0.

property **quirks**: *List[str]*

Quirks specific to this labware.

New in version 2.0.

reset(*self*) → `'None'`

Reset all tips in a tip rack.

Changed in version 2.14: This method will raise an exception if you call it on a labware that isn't a tip rack. Formerly, it would do nothing.

New in version 2.0.

rows(*self*, **args: 'Union[int, str]'*) → `'List[List[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, 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[Well]]'`

Deprecated since version 2.0: Use `rows_by_name()` instead.

New in version 2.0.

rows_by_name(*self*) → `'Dict[str, List[Well]]'`

Accessor function used to navigate through a labware by row name.

Returns:: Dictionary of well lists keyed by row name

New in version 2.0.

set_calibration(*self*, *delta: 'Point'*) → 'None'

An internal, deprecated method used for updating the offset on the object.

Deprecated since version 2.14.

set_offset(*self*, *x: 'float'*, *y: 'float'*, *z: 'float'*) → 'None'

Set the labware's position offset.

The offset is an x, y, z vector in deck coordinates (see protocol-api-deck-coords) that the motion system will add to any movement targeting this labware instance.

The offset will *not* apply to any other labware instances, even if those labware are of the same type.

> **Caution:**
>
> This method is *only* for use with mechanisms like `opentrons.execute.get_protocol_api`, which lack an interactive way to adjust labware offsets. (See Advanced Control.)
>
> If you're uploading a protocol via the Opentrons App, don't use this method, because it will produce undefined behavior. Instead, use Labware Position Check in the app.
>
> Because protocols using API version 2.14 or higher can currently *only* be uploaded via the Opentrons App, it doesn't make sense to use this method with them. Trying to do so will raise an exception.

New in version 2.12.

property **tip_length**: *float*

New in version 2.0.

property **uri**: *str*

A string fully identifying the labware.

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

New in version 2.0.

well(*self*, *idx: 'Union[int, str]'*) → 'Well'

Deprecated—use result of *wells* or *wells_by_name*

New in version 2.0.

wells(*self*, **args: 'Union[str, int]'*) → 'List[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, 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, Well]'

Deprecated since version 2.0: Use `wells_by_name()` or dict access instead.

New in version 2.0.

wells_by_name(*self*) → 'Dict[str, Well]'

Accessor function used to create a look-up table of Wells by name.

Returns:: Dictionary of well objects keyed by well name

New in version 2.0.

class opentrons.protocol_api.**Well**(*parent: Labware, core: WellCore, api_version: APIVersion*)

The Well class represents a single well in a `Labware`. It provides parameters and functions for three major uses:

- Calculating positions relative to the well. See Position Relative to Labware for details.
- Returning well measurements. see Well Dimensions for details.
- Specifying what liquid should be in the well at the beginning of a protocol. See Labeling Liquids in Wells for details.

property **api_version**: *APIVersion*

New in version 2.0.

bottom(*self, z: 'float' = 0.0*) → 'Location'

Parameters:: **z** – An offset on the z-axis, in mm. Positive offsets are higher and negative offsets are lower.

Returns:: A `Location` corresponding to the absolute position of the bottom-center of the well, plus the `z` offset (if specified).

New in version 2.0.

center(*self*) → 'Location'

Returns:: A `Location` corresponding to the absolute position of the center of the well (in all three dimensions).

New in version 2.0.

property **depth**: *float*

The depth, in mm, of a well along the z-axis, from the very top of the well to the very bottom.

New in version 2.9.

property **diameter**: *Optional[float]*

The diameter, in mm, of a circular well. Returns `None` if the well is not circular.

New in version 2.0.

from_center_cartesian(*self, x: 'float', y: 'float', z: 'float'*) → 'Point'

Specifies a `Point` based on fractions of the distance from the center of the well to the edge along each axis.

For example, `from_center_cartesian(0, 0, 0.5)` specifies a point at the well's center on the x- and y-axis, and half of the distance from the center of the well to its top along the z-axis. To move the pipette to that location, construct a `Location` relative to the same well:

```
location = types.Location(
    plate["A1"].from_center_cartesian(0, 0, 0.5), plate["A1"]
)
pipette.move_to(location)
```

See Points and Locations for more information.

Parameters::
- **x** – The fraction of the distance from the well's center to its edge along the x-axis. Negative values are to the left, and positive values are to the right.
- **y** – The fraction of the distance from the well's center to its edge along the y-axis. Negative values are to the front, and positive values are to the back.
- **z** – The fraction of the distance from the well's center to its edge along the x-axis. Negative values are down, and positive values are up.

Returns:: A `Point` representing the specified position in absolute deck coordinates.

Note:

Even if the absolute values of `x`, `y`, and `z` are all less than 1, a location constructed from the well and the result of `from_center_cartesian` may be outside of the physical well. For example, `from_center_cartesian(0.9, 0.9, 0)` would be outside of a cylindrical well, but inside a square well.

Whether this well contains a tip. Always False if the parent labware isn't a tip rack.

New in version 2.0.

property **length**: *Optional[float]*

The length, in mm, of a rectangular well along the x-axis (left to right). Returns `None` if the well is not rectangular.

New in version 2.9.

load_liquid(*self*, *liquid: 'Liquid'*, *volume: 'float'*) → 'None'

Load a liquid into a well.

Parameters::
- **liquid** (*Liquid*) – The liquid to load into the well.
- **volume** (*float*) – The volume of liquid to load, in µL.

New in version 2.14.

property **parent**: *Labware*

New in version 2.0.

top(*self*, *z: 'float' = 0.0*) → 'Location'

Parameters:: **z** – An offset on the z-axis, in mm. Positive offsets are higher and negative offsets are lower.

Returns:: A **Location** corresponding to the absolute position of the top-center of the well, plus the `z` offset (if specified).

New in version 2.0.

property **well_name**: *str*

New in version 2.7.

property **width**: *Optional[float]*

The width, in mm, of a rectangular well along the y-axis (front to back). Returns `None` if the well is not rectangular.

New in version 2.9.

Modules

class opentrons.protocol_api.**HeaterShakerContext**(*core: AbstractModuleCore*, *protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore]*, *core_map: LoadedCoreMap*, *api_version: APIVersion*, *broker: LegacyBroker*)

An object representing a connected Heater-Shaker Module.

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

New in version 2.13.

property **api_version**: *APIVersion*

New in version 2.0.

close_labware_latch(*self*) → 'None'

Closes the labware latch.

The labware latch needs to be closed using this method before sending a shake command, even if the latch was manually closed before starting the protocol.

New in version 2.13.

property **current_speed**: *int*

The current speed of the Heater-Shaker's plate in rpm.

New in version 2.13.

property **current_temperature**: *float*

The current temperature of the Heater-Shaker's plate in °C.

Returns `23` in simulation if no target temperature has been set.

New in version 2.13.

The module will passively cool to room temperature. The Heater-Shaker does not have active cooling.

New in version 2.13.

deactivate_shaker(*self*) → 'None'

Stops shaking.

Decelerating to 0 rpm typically only takes a few seconds.

New in version 2.13.

property **labware**: *Optional[Labware]*

The labware (if any) present on this module.

New in version 2.0.

property **labware_latch_status**: *str*

One of six possible latch statuses:

- `opening` – The latch is currently opening (in motion).
- `idle_open` – The latch is open and not moving.
- `closing` – The latch is currently closing (in motion).
- `idle_closed` – The latch is closed and not moving.
- `idle_unknown` – The default status upon reset, regardless of physical latch position. Use **close_labware_latch()** before other commands requiring confirmation that the latch is closed.
- `unknown` – The latch status can't be determined.

New in version 2.13.

load_adapter(*self*, *name: 'str'*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Load an adapter onto the module using its load parameters.

The parameters of this function behave like those of **ProtocolContext.load_adapter** (which loads adapters directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded adapter object.

New in version 2.15.

load_adapter_from_definition(*self*, *definition: 'LabwareDefinition'*) → 'Labware'

Load an adapter onto the module using an inline definition.

Parameters:: **definition** – The labware definition.

Returns:: The initialized and loaded labware object.

New in version 2.15.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → 'Labware'

Load a labware onto the module using its load parameters.

The parameters of this function behave like those of **ProtocolContext.load_labware** (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

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

load_labware_by_name(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Deprecated since version 2.0: Use **load_labware()** instead.

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → 'Labware'

Load a labware onto the module using an inline definition.

Returns:: The initialized and loaded labware object.

New in version 2.0.

property **model**: *Union[typing_extensions.Literal[magneticModuleV1, magneticModuleV2], typing_extensions.Literal[temperatureModuleV1, temperatureModuleV2], typing_extensions.Literal[thermocyclerModuleV1, thermocyclerModuleV2], typing_extensions.Literal[heaterShakerModuleV1], typing_extensions.Literal[magneticBlockV1]]*

Get the module's model identifier.

New in version 2.14.

open_labware_latch(*self*) → 'None'

Open the Heater-Shaker's labware latch.

The labware latch needs to be closed before:

- Shaking
- Pipetting to or from the labware on the Heater-Shaker
- Pipetting to or from labware to the left or right of the Heater-Shaker

Attempting to open the latch while the Heater-Shaker is shaking will raise an error.

> **Note:**
>
> Before opening the latch, this command will retract the pipettes upward if they are parked adjacent to the left or right of the Heater-Shaker.

New in version 2.13.

property **parent**: *str*

The name of the slot the module is on.

On a Flex, this will be like `"D1"`. On an OT-2, this will be like `"1"`. See Deck Slots.

New in version 2.14.

property **serial_number**: *str*

Get the module's unique hardware serial number.

New in version 2.14.

set_and_wait_for_shake_speed(*self*, *rpm: 'int'*) → 'None'

Set a shake speed in rpm and block execution of further commands until the module reaches the target.

Reaching a target shake speed typically only takes a few seconds.

> **Note:**
>
> Before shaking, this command will retract the pipettes upward if they are parked adjacent to the Heater-Shaker.

Parameters:: **rpm** – A value between 200 and 3000, representing the target shake speed in revolutions per minute.

New in version 2.13.

set_and_wait_for_temperature(*self*, *celsius: 'float'*) → 'None'

Set a target temperature and wait until the module reaches the target.

No other protocol commands will execute while waiting for the temperature.

Parameters:: **celsius** – A value between 27 and 95, representing the target temperature in °C. Values are automatically truncated to two decimal places, and the Heater-Shaker module has a temperature accuracy of ±0.5 °C.

Set target temperature and return immediately.

Sets the Heater-Shaker's target temperature and returns immediately without waiting for the target to be reached. Does not delay the protocol until target temperature has reached. Use `wait_for_temperature()` to delay protocol execution.

Parameters:: **celsius** – A value between 27 and 95, representing the target temperature in °C. Values are automatically truncated to two decimal places, and the Heater-Shaker module has a temperature accuracy of ±0.5 °C.

New in version 2.13.

property **speed_status**: *str*

One of five possible shaking statuses:

- `holding at target` – The module has reached its target shake speed and is actively maintaining that speed.
- `speeding up` – The module is increasing its shake speed towards a target.
- `slowing down` – The module was previously shaking at a faster speed and is currently reducing its speed to a lower target or to deactivate.
- `idle` – The module is not shaking.
- `error` – The shaking status can't be determined.

New in version 2.13.

property **target_speed**: *Optional[int]*

Target speed of the Heater-Shaker's plate in rpm.

New in version 2.13.

property **target_temperature**: *Optional[float]*

The target temperature of the Heater-Shaker's plate in °C.

Returns `None` if no target has been set.

New in version 2.13.

property **temperature_status**: *str*

One of five possible temperature statuses:

- `holding at target` – The module has reached its target temperature and is actively maintaining that temperature.
- `cooling` – The module has previously heated and is now passively cooling. *The Heater-Shaker does not have active cooling.*
- `heating` – The module is heating to a target temperature.
- `idle` – The module has not heated since the beginning of the protocol.
- `error` – The temperature status can't be determined.

New in version 2.13.

property **type**: *Union[typing_extensions.Literal[magneticModuleType], typing_extensions.Literal[temperatureModuleType], typing_extensions.Literal[thermocyclerModuleType], typing_extensions.Literal[heaterShakerModuleType], typing_extensions.Literal[magneticBlockType]]*

Get the module's general type identifier.

New in version 2.14.

wait_for_temperature(*self*) → 'None'

Delays protocol execution until the Heater-Shaker has reached its target temperature.

Raises an error if no target temperature was previously set.

New in version 2.13.

class opentrons.protocol_api.**MagneticBlockContext**(*core: AbstractModuleCore, protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore], core_map: LoadedCoreMap, api_version: APIVersion, broker: LegacyBroker*)

An object representing a Magnetic Block.

property **api_version**: *APIVersion*

New in version 2.0.

property **labware**: *Optional[Labware]*

The labware (if any) present on this module.

New in version 2.0.

load_adapter(*self*, *name: 'str'*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → *'Labware'*

Load an adapter onto the module using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_adapter` (which loads adapters directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded adapter object.

New in version 2.15.

load_adapter_from_definition(*self*, *definition: 'LabwareDefinition'*) → *'Labware'*

Load an adapter onto the module using an inline definition.

Parameters:: **definition** – The labware definition.

Returns:: The initialized and loaded labware object.

New in version 2.15.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → *'Labware'*

Load a labware onto the module using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_labware` (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

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

load_labware_by_name(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → *'Labware'*

Deprecated since version 2.0: Use `load_labware()` instead.

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → *'Labware'*

Load a labware onto 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.

property **model**: *Union[typing_extensions.Literal[magneticModuleV1, magneticModuleV2], typing_extensions.Literal[temperatureModuleV1, temperatureModuleV2], typing_extensions.Literal[thermocyclerModuleV1, thermocyclerModuleV2], typing_extensions.Literal[heaterShakerModuleV1], typing_extensions.Literal[magneticBlockV1]]*

Get the module's model identifier.

New in version 2.14.

property **parent**: *str*

The name of the slot the module is on.

On a Flex, this will be like `"D1"`. On an OT-2, this will be like `"1"`. See Deck Slots.

typing_extensions.Literal[heaterShakerModuleType], typing_extensions.Literal[magneticBlockType]]

Get the module's general type identifier.

New in version 2.14.

class opentrons.protocol_api.**MagneticModuleContext**(*core: AbstractModuleCore, protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore], core_map: LoadedCoreMap, api_version: APIVersion, broker: LegacyBroker*)

An object representing a connected Magnetic Module.

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

New in version 2.0.

property **api_version**: *APIVersion*

New in version 2.0.

disengage(*self*) → 'None'

Lower the magnets back into the Magnetic Module.

New in version 2.0.

engage(*self, height: 'Optional[float]' = None, offset: 'Optional[float]' = None, height_from_base: 'Optional[float]' = None*) → 'None'

Raise the Magnetic Module's magnets. You can specify how high the magnets should move:

- No parameter: Move to the default height for the loaded labware. If the loaded labware has no default, or if no labware is loaded, this will raise an error.
- `height_from_base` – Move this many millimeters above the bottom of the labware. Acceptable values are between `0` and `25`.

  This is the recommended way to adjust the magnets' height.

  New in version 2.2.
- `offset` – Move this many millimeters above (positive value) or below (negative value) the default height for the loaded labware. The sum of the default height and `offset` must be between 0 and 25.
- `height` – Intended to move this many millimeters above the magnets' home position. However, depending on the generation of module and the loaded labware, this may produce unpredictable results. You should normally use `height_from_base` instead.

  Changed in version 2.14: This parameter has been removed.

You shouldn't specify more than one of these parameters. However, if you do, their order of precedence is `height`, then `height_from_base`, then `offset`.

New in version 2.0.

property **labware**: *Optional[Labware]*

The labware (if any) present on this module.

New in version 2.0.

load_adapter(*self, name: 'str', namespace: 'Optional[str]' = None, version: 'Optional[int]' = None*) → 'Labware'

Load an adapter onto the module using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_adapter` (which loads adapters directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded adapter object.

New in version 2.15.

load_adapter_from_definition(*self, definition: 'LabwareDefinition'*) → 'Labware'

Load an adapter onto the module using an inline definition.

New in version 2.15.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → 'Labware'

Load a labware onto the module using its load parameters.

The parameters of this function behave like those of **ProtocolContext.load_labware** (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

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

load_labware_by_name(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Deprecated since version 2.0: Use **load_labware()** instead.

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → 'Labware'

Load a labware onto 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.

property **model**: *Union[typing_extensions.Literal[magneticModuleV1, magneticModuleV2], typing_extensions.Literal[temperatureModuleV1, temperatureModuleV2], typing_extensions.Literal[thermocyclerModuleV1, thermocyclerModuleV2], typing_extensions.Literal[heaterShakerModuleV1], typing_extensions.Literal[magneticBlockV1]]*

Get the module's model identifier.

New in version 2.14.

property **parent**: *str*

The name of the slot the module is on.

On a Flex, this will be like `"D1"`. On an OT-2, this will be like `"1"`. See Deck Slots.

New in version 2.14.

property **serial_number**: *str*

Get the module's unique hardware serial number.

New in version 2.14.

property **status**: *str*

The status of the module, either `engaged` or `disengaged`.

New in version 2.0.

property **type**: *Union[typing_extensions.Literal[magneticModuleType], typing_extensions.Literal[temperatureModuleType], typing_extensions.Literal[thermocyclerModuleType], typing_extensions.Literal[heaterShakerModuleType], typing_extensions.Literal[magneticBlockType]]*

Get the module's general type identifier.

New in version 2.14.

class opentrons.protocol_api.**TemperatureModuleContext**(*core: AbstractModuleCore*, *protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore]*, *core_map: LoadedCoreMap*, *api_version: APIVersion*, *broker: LegacyBroker*)

An object representing a connected Temperature Module.

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

New in version 2.0.

deactivate(*self*) → 'None'

Stop heating or cooling, and turn off the fan.

New in version 2.0.

property **labware**: *Optional[Labware]*

The labware (if any) present on this module.

New in version 2.0.

load_adapter(*self*, *name: 'str'*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Load an adapter onto the module using its load parameters.

The parameters of this function behave like those of **ProtocolContext.load_adapter** (which loads adapters directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded adapter object.

New in version 2.15.

load_adapter_from_definition(*self*, *definition: 'LabwareDefinition'*) → 'Labware'

Load an adapter onto the module using an inline definition.

Parameters:: **definition** – The labware definition.

Returns:: The initialized and loaded labware object.

New in version 2.15.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → 'Labware'

Load a labware onto the module using its load parameters.

The parameters of this function behave like those of **ProtocolContext.load_labware** (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

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

load_labware_by_name(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Deprecated since version 2.0: Use **load_labware()** instead.

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → 'Labware'

Load a labware onto 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.

property **model**: *Union[typing_extensions.Literal[magneticModuleV1, magneticModuleV2], typing_extensions.Literal[temperatureModuleV1, temperatureModuleV2], typing_extensions.Literal[thermocyclerModuleV1, thermocyclerModuleV2], typing_extensions.Literal[heaterShakerModuleV1], typing_extensions.Literal[magneticBlockV1]]*

Get the module's model identifier.

New in version 2.14.

property **parent**: *str*

The name of the slot the module is on.

property **serial_number**: *str*

Get the module's unique hardware serial number.

New in version 2.14.

set_temperature(*self*, *celsius: 'float'*) → 'None'

Set a target temperature and wait until the module reaches the target.

No other protocol commands will execute while waiting for the temperature.

Parameters:: **celsius** – A value between 4 and 95, representing the target temperature in °C.

New in version 2.0.

property **status**: *str*

One of four possible temperature statuses:

- `holding at target` – The module has reached its target temperature and is actively maintaining that temperature.
- `cooling` – The module is cooling to a target temperature.
- `heating` – The module is heating to a target temperature.
- `idle` – The module has been deactivated.

New in version 2.3.

property **target**: *Optional[float]*

The target temperature of the Temperature Module's deck in °C.

Returns `None` if no target has been set.

New in version 2.0.

property **temperature**: *float*

The current temperature of the Temperature Module's deck in °C.

Returns `0` in simulation if no target temperature has been set.

New in version 2.0.

property **type**: *Union[typing_extensions.Literal[magneticModuleType], typing_extensions.Literal[temperatureModuleType], typing_extensions.Literal[thermocyclerModuleType], typing_extensions.Literal[heaterShakerModuleType], typing_extensions.Literal[magneticBlockType]]*

Get the module's general type identifier.

New in version 2.14.

class opentrons.protocol_api.**ThermocyclerContext**(*core: AbstractModuleCore*, *protocol_core: AbstractProtocol[AbstractInstrument[AbstractWellCore], AbstractLabware[AbstractWellCore], AbstractModuleCore]*, *core_map: LoadedCoreMap*, *api_version: APIVersion*, *broker: LegacyBroker*)

An object representing a connected Thermocycler Module.

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

New in version 2.0.

property **api_version**: *APIVersion*

New in version 2.0.

property **block_target_temperature**: *Optional[float]*

The target temperature of the well block in °C.

New in version 2.0.

property **block_temperature**: *Optional[float]*

The current temperature of the well block in °C.

New in version 2.0.

property **block_temperature_status**: *str*

temperature.

- `cooling` – The block is cooling to a target temperature.
- `heating` – The block is heating to a target temperature.
- `idle` – The block is not currently heating or cooling.
- `error` – The temperature status can't be determined.

New in version 2.0.

close_lid(*self*) → 'str'

Close the lid.

New in version 2.0.

deactivate(*self*) → 'None'

Turn off both the well block temperature controller and the lid heater.

New in version 2.0.

deactivate_block(*self*) → 'None'

Turn off the well block temperature controller.

New in version 2.0.

deactivate_lid(*self*) → 'None'

Turn off the lid heater.

New in version 2.0.

execute_profile(*self*, *steps: 'List[ThermocyclerStep]'*, *repetitions: 'int'*, *block_max_volume: 'Optional[float]'* = *None*) → 'None'

Execute a Thermocycler profile, defined as a cycle of `steps`, for a given number of `repetitions`.

Parameters::

- **steps** – List of unique steps that make up a single cycle. Each list item should be a dictionary that maps to the parameters of the **`set_block_temperature()`** method with a `temperature` key, and either or both of `hold_time_seconds` and `hold_time_minutes`.
- **repetitions** – The number of times to repeat the cycled steps.
- **block_max_volume** – The greatest volume of liquid contained in any individual well of the loaded labware, in µL. If not specified, the default is 25 µL.

New in version 2.0.

property **labware**: *Optional[Labware]*

The labware (if any) present on this module.

New in version 2.0.

property **lid_position**: *Optional[str]*

One of these possible lid statuses:

- `closed` – The lid is closed.
- `in_between` – The lid is neither open nor closed.
- `open` – The lid is open.
- `unknown` – The lid position can't be determined.

New in version 2.0.

property **lid_target_temperature**: *Optional[float]*

The target temperature of the lid in °C.

New in version 2.0.

property **lid_temperature**: *Optional[float]*

The current temperature of the lid in °C.

New in version 2.0.

property **lid_temperature_status**: *Optional[str]*

The Thermocycler lid does not have active cooling.

- `heating` – The lid is heating to a target temperature.
- `idle` – The lid has not heated since the beginning of the protocol.
- `error` – The temperature status can't be determined.

New in version 2.0.

load_adapter(*self*, *name: 'str'*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Load an adapter onto the module using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_adapter` (which loads adapters directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded adapter object.

New in version 2.15.

load_adapter_from_definition(*self*, *definition: 'LabwareDefinition'*) → 'Labware'

Load an adapter onto the module using an inline definition.

Parameters:: **definition** – The labware definition.

Returns:: The initialized and loaded labware object.

New in version 2.15.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*, *adapter: 'Optional[str]' = None*) → 'Labware'

Load a labware onto the module using its load parameters.

The parameters of this function behave like those of `ProtocolContext.load_labware` (which loads labware directly onto the deck). Note that the parameter `name` here corresponds to `load_name` on the `ProtocolContext` function.

Returns:: The initialized and loaded labware object.

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

load_labware_by_name(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'Optional[int]' = None*) → 'Labware'

Deprecated since version 2.0: Use `load_labware()` instead.

New in version 2.1.

load_labware_from_definition(*self*, *definition: 'LabwareDefinition'*, *label: 'Optional[str]' = None*) → 'Labware'

Load a labware onto 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.

property **model**: *Union[typing_extensions.Literal[magneticModuleV1, magneticModuleV2], typing_extensions.Literal[temperatureModuleV1, temperatureModuleV2], typing_extensions.Literal[thermocyclerModuleV1, thermocyclerModuleV2], typing_extensions.Literal[heaterShakerModuleV1], typing_extensions.Literal[magneticBlockV1]]*

Get the module's model identifier.

New in version 2.14.

open_lid(*self*) → 'str'

Open the lid.

New in version 2.0.

New in version 2.14.

property **serial_number**: *str*

Get the module's unique hardware serial number.

New in version 2.14.

set_block_temperature(*self*, *temperature: 'float'*, *hold_time_seconds: 'Optional[float]' = None*, *hold_time_minutes: 'Optional[float]' = None*, *ramp_rate: 'Optional[float]' = None*, *block_max_volume: 'Optional[float]' = None*) → 'None'

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

Parameters::
- **temperature** – A value between 4 and 99, representing the target temperature in °C.
- **hold_time_minutes** – The number of minutes to hold, after reaching `temperature`, before proceeding to the next command. If `hold_time_seconds` is also specified, the times are added together.
- **hold_time_seconds** – The number of seconds to hold, after reaching `temperature`, before proceeding to the next command. If `hold_time_minutes` is also specified, the times are added together.
- **block_max_volume** – The greatest volume of liquid contained in any individual well of the loaded labware, in µL. If not specified, the default is 25 µL.

New in version 2.0.

set_lid_temperature(*self*, *temperature: 'float'*) → 'None'

Set the target temperature for the heated lid, in °C.

Parameters:: **temperature** – A value between 37 and 110, representing the target temperature in °C.

New in version 2.0.

property **type**: *Union[typing_extensions.Literal[magneticModuleType], typing_extensions.Literal[temperatureModuleType], typing_extensions.Literal[thermocyclerModuleType], typing_extensions.Literal[heaterShakerModuleType], typing_extensions.Literal[magneticBlockType]]*

Get the module's general type identifier.

New in version 2.14.

Useful Types and Definitions

class opentrons.types.**Location**(*point: Point*, *labware: Union[Labware, Well, str, ModuleGeometry, LabwareLike, None, ModuleContext]*)

A location to target as a motion.

The location contains a **`Point`** (in protocol-api-deck-coords) and possibly an associated **`Labware`** or **`Well`** instance.

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

Warning:

The `.labware` 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` attribute.

Warning:

The `==` operation compares both the position and associated labware. If you only need to compare locations, compare the `.point` of each item.

move(*self*, *point: 'Point'*) → "'Location'"

```
>>> loc = Location(Point(1, 1, 1), None)
>>> new_loc = loc.move(Point(1, 1, 1))
>>>
>>> # The new point is the old one plus the given offset.
>>> assert new_loc.point == Point(2, 2, 2)  # True
>>>
>>> # The old point hasn't changed.
>>> assert loc.point == Point(1, 1, 1)  # True
```

class opentrons.types.**Mount**(*value*)

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

opentrons.protocol_api.**OFF_DECK**

A special location value, indicating that a labware is not currently on the robot's deck.

See The Off-Deck Location for details on using `OFF_DECK` with **`ProtocolContext.move_labware()`**.

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: Union[BinaryIO, TextIO]*, *protocol_name: str*, *propagate_logs: bool = False*, *log_level: str = 'warning'*, *emit_runlog: Union[Callable[[Union[opentrons.commands.types.DropTipMessage, opentrons.commands.types.PickUpTipMessage, opentrons.commands.types.ReturnTipMessage, opentrons.commands.types.AirGapMessage, opentrons.commands.types.TouchTipMessage, opentrons.commands.types.BlowOutMessage, opentrons.commands.types.MixMessage, opentrons.commands.types.TransferMessage, opentrons.commands.types.DistributeMessage, opentrons.commands.types.ConsolidateMessage, opentrons.commands.types.DispenseMessage, opentrons.commands.types.AspirateMessage, opentrons.commands.types.HomeMessage, opentrons.commands.types.HeaterShakerSetTargetTemperatureMessage, opentrons.commands.types.HeaterShakerWaitForTemperatureMessage, opentrons.commands.types.HeaterShakerSetAndWaitForShakeSpeedMessage, opentrons.commands.types.HeaterShakerOpenLabwareLatchMessage, opentrons.commands.types.HeaterShakerCloseLabwareLatchMessage, opentrons.commands.types.HeaterShakerDeactivateShakerMessage, opentrons.commands.types.HeaterShakerDeactivateHeaterMessage, opentrons.commands.types.ThermocyclerCloseMessage, opentrons.commands.types.ThermocyclerWaitForLidTempMessage, opentrons.commands.types.ThermocyclerDeactivateMessage, opentrons.commands.types.ThermocyclerDeactivateBlockMessage, opentrons.commands.types.ThermocyclerDeactivateLidMessage, opentrons.commands.types.ThermocyclerSetLidTempMessage, opentrons.commands.types.ThermocyclerWaitForTempMessage, opentrons.commands.types.ThermocyclerWaitForHoldMessage, opentrons.commands.types.ThermocyclerExecuteProfileMessage, opentrons.commands.types.ThermocyclerSetBlockTempMessage, opentrons.commands.types.ThermocyclerOpenMessage, opentrons.commands.types.TempdeckSetTempMessage, opentrons.commands.types.TempdeckDeactivateMessage, opentrons.commands.types.MagdeckEngageMessage, opentrons.commands.types.MagdeckDisengageMessage, opentrons.commands.types.MagdeckCalibrateMessage, opentrons.commands.types.CommentMessage, opentrons.commands.types.DelayMessage, opentrons.commands.types.PauseMessage, opentrons.commands.types.ResumeMessage, opentrons.commands.types.MoveToMessage]], NoneType], NoneType] = None*, *custom_labware_paths: Union[List[str], NoneType] = None*, *custom_data_paths: Union[List[str], NoneType] = None*) → None

Run the protocol itself.

This is a one-stop function to run 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 call from the command line use either the autogenerated entrypoint `opentrons_execute` or `python -m opentrons.execute`.

Parameters::

- **protocol_file** – The protocol file to execute
- **protocol_name** – 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** – 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** – The level of logs to emit on the command line: `"debug"`, `"info"`, `"warning"`, or `"error"`. Defaults to `"warning"`.
- **emit_runlog** –
  A callback for printing the run log. If specified, this will be called whenever a command adds an entry to the run log, 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:

```
{
  'name': command_name,
  'payload': {
    'text': string_command_text,
    # The rest of this struct is
    # command-dependent; see
    # opentrons.commands.commands.
   }
}
```

  Note:
  In older software versions, `payload["text"]` was a format string. To get human-readable text, you had to do `payload["text"].format(**payload)`. Don't do that anymore. If `payload["text"]` happens to contain any `{` or `}` characters, it can confuse `.format()` and cause it to raise a `KeyError`.

- **custom_labware_paths** – A list of directories to search for custom labware. Loads valid labware from these paths and makes them available to the protocol context. If this is `None` (the default), and this function is called on a robot, it will look in the `labware` subdirectory of the Jupyter data directory.
- **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`.

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.api_support.types.APIVersion], bundled_labware: Union[Dict[str, ForwardRef('LabwareDefinitionDict')], NoneType] = None, bundled_data: Union[Dict[str, bytes], NoneType] = None, extra_labware: Union[Dict[str, ForwardRef('LabwareDefinitionDict')], NoneType] = None*) → opentrons.protocol_api.protocol_context.ProtocolContext

Build and return a `protocol_api.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()
```

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 **opentrons.protocol_api.ProtocolContext.bundled_data**.
- **extra_labware** – A mapping from labware load names to custom labware definitions. If this is `None` (the default), and this function is called on a robot, it will look for labware in the `labware` subdirectory of the Jupyter data directory.

Returns:: 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.

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**

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.api_support.types.APIVersion], bundled_labware: Union[Dict[str, ForwardRef('LabwareDefinitionDict')], NoneType] = None, bundled_data: Union[Dict[str, bytes], NoneType] = None, extra_labware: Union[Dict[str, ForwardRef('LabwareDefinitionDict')], NoneType] = None, hardware_simulator: Union[opentrons.hardware_control.thread_manager.ThreadManager[Union[opentrons.hardware_control.protocols.HardwareControlInterface[opentr opentrons.hardware_control.protocols.HardwareControlInterface[opentrons.hardware_control.ot3_calibration.OT3Transforms]]], NoneType] = None, *, robot_type: Union[typing_extensions.Literal['OT-2', 'Flex'], NoneType] = None*) → opentrons.protocol_api.protocol_context.ProtocolContext

Build and return a `protocol_api.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()
```

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

lowed 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 `opentrons.protocol_api.ProtocolContext.bundled_data`.
- **extra_labware** – A mapping from labware load names to custom labware definitions. If this is `None` (the default), and this function is called on a robot, it will look for labware in the `labware` subdirectory of the Jupyter data directory.
- **hardware_simulator** – If specified, a hardware simulator instance.
- **robot_type** – The type of robot to simulate: either `"Flex"` or `"OT-2"`. If you're running this function on a robot, the default is the type of that robot. Otherwise, the default is `"OT-2"`, for backwards compatibility.

Returns:: The protocol context.

opentrons.simulate.**main**() → int

Run the simulation

opentrons.simulate.**simulate**(*protocol_file: Union[BinaryIO, TextIO]*, *file_name: Union[str, NoneType] = None*, *custom_labware_paths: Union[List[str], NoneType] = None*, *custom_data_paths: Union[List[str], NoneType] = None*, *propagate_logs: bool = False*, *hardware_simulator_file_path: Union[str, NoneType] = None*, *duration_estimator: Union[opentrons.protocols.duration.estimator.DurationEstimator, NoneType] = 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. The human-readable run log text is available at `payload["text"]`. The other keys of `payload` are command-dependent; see `opentrons.commands`.

  Note:

  In older software versions, `payload["text"]` was a format string. To get human-readable text, you had to do `payload["text"].format(**payload)`. Don't do that anymore. If `payload["text"]` happens to contain any `{` or `}` characters, it can confuse `.format()` and cause it to raise a `KeyError`.

- `logs`: Any log messages that occurred during execution of this command, as a standard Python `LogRecord`.

Parameters::

- **protocol_file** – The protocol file to simulate.
- **file_name** – The name of the file
- **custom_labware_paths** – A list of directories to search for custom labware. Loads valid labware from these paths and makes them available to the protocol context. If this is `None` (the default), and this function is called on a robot, it will look in the `labware` subdirectory of the Jupyter data directory.
- **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 `protocol_api.ProtocolContext.bundled_data`.
- **hardware_simulator_file_path** – A path to a JSON file defining a hardware simulator.

but most logs that occur during protocol simulation are best associated with the actions in the protocol that cause them. Default: `False`

- **log_level** – The level of logs to capture in the run log: `"debug"`, `"info"`, `"warning"`, or `"error"`. Defaults to `"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 and this is an unbundled Protocol API v2 python protocol. In other cases it is None.

Sign Up For Our Newsletter

Email**

SUBMIT

© OPENTRONS 2023