Introduction to Opentrons OT-2 for AI Assistants

The Opentrons OT-2 is an open-source liquid handling robot designed for automated pipetting and protocol execution in life science laboratories. This document provides a concise overview of the key information needed to control an OT-2 robot using an AI language model.
The goal is to enable AI assistants to understand OT-2 capabilities, constraints, and API commands in order to generate valid OT-2 protocols based on high-level user intent. The document covers:

OT-2 hardware overview and specifications

- API basics and key concepts
- Pipettes and liquid handling
- Labware and deck setup
- Building block commands for aspirate, dispense, etc.
- Complex commands for common pipetting patterns
- Protocol examples demonstrating API usage

By digesting this curated information, language models can engage in intelligent conversational protocol generation, guiding users from sample descriptions to executable Python code that runs on the OT-2.

https://docs.opentrons.com/v2/

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

Similar to the code sample in How the API Works, here's how you use the ProtocolContext.load_labware() method to load labware on either Flex or OT-2.

```
#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"
)
```
Loading labware this way prevents you from moving the labware onto or off of the adapter, so it’s less flexible than loading the two separately. Avoid using combination definitions unless your protocol specifies an apiLevel of 2.14 or lower.

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.

_images/Well_Iteration.png

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
The simplest way to refer to a single well is by its well_name, like A1 or D6. Referencing a particular key in the result of Labware.wells_by_name() accomplishes this. This is such a common task that the API also has an equivalent shortcut: dictionary indexing.

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

Optionally, you can specify the liquids that should be in various wells at the beginning of your protocol. Doing so helps you identify well contents by name and volume, and adds corresponding labels to a single well, or group of wells, in well plates and reservoirs. You can view the initial liquid setup:

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.

Labeling vs Handling Liquids

The load_liquid arguments include a volume amount (volume=n in µL). This amount is just a label. It isn't a command or 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
```

Hardware modules

Temperature controller

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

The Temperature Module is represented in code by a TemperatureModuleContext object, which has methods for setting target temperatures and reading the module's status. This example demonstrates loading a Temperature Module GEN2 and loading a well plate on top of it.

```
temp_mod = protocol.load_module(
    module_name="temperature module gen2", location="D3"
)
```

New in version 2.3.

Loading Labware
Use the Temperature Module's load_adapter() and load_labware() methods to specify what you will place on the module. You may use one or both of the methods, depending on the labware you're using. 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 Temperature Module ready to use right out of the box.

Standalone Adapters
You can use these standalone adapter definitions to load Opentrons verified or custom labware on top of the Temperature Module.

Adapter Type

API Load Name

Opentrons Aluminum Flat Bottom Plate

opentrons_aluminum_flat_bottom_plate

Opentrons 96 Well Aluminum Block

opentrons_96_well_aluminum_block

For example, these commands load a PCR plate on top of the 96-well block:

```
temp_adapter = temp_mod.load_adapter(
    "opentrons_96_well_aluminum_block"
)
temp_plate = temp_adapter.load_labware(
    "nest_96_wellplate_100ul_pcr_full_skirt"
)
```

New in version 2.15: The load_adapter() method.

Note
You can also load labware directly onto the Temperature Module. In API version 2.14 and earlier, this was the correct way to load labware on top of the flat bottom plate. In API version 2.15 and later, you should load both the adapter and the labware with separate commands.

Block-and-tube combinations
You can use these combination labware definitions to load various types of tubes into the 24-well thermal block on top of the Temperature Module. There is no standalone definition for the 24-well block.

Tube Type

API Load Name

Generic 2 mL screw cap

opentrons_24_aluminumblock_generic_2ml_screwcap

NEST 0.5 mL screw cap

opentrons_24_aluminumblock_nest_0.5ml_screwcap

NEST 1.5 mL screw cap

opentrons_24_aluminumblock_nest_1.5ml_screwcap

NEST 1.5 mL snap cap

opentrons_24_aluminumblock_nest_1.5ml_snapcap

NEST 2 mL screw cap

opentrons_24_aluminumblock_nest_2ml_screwcap

NEST 2 mL snap cap

opentrons_24_aluminumblock_nest_2ml_snapcap

For example, this command loads the 24-well block with generic 2 mL tubes:

```
temp_tubes = temp_mod.load_labware(
    "opentrons_24_aluminumblock_generic_2ml_screwcap"
)
```

New in version 2.0.

Block-and-plate combinations

The Temperature Module supports these 96-well block and labware combinations for backwards compatibility. If your protocol specifies an apiLevel of 2.15 or higher, you should use the standalone 96-well block definition instead.

96-well block contents

API Load Name

Bio-Rad well plate 200 µL

opentrons_96_aluminumblock_biorad_wellplate_200uL

Generic PCR strip 200 µL

opentrons_96_aluminumblock_generic_pcr_strip_200uL

NEST well plate 100 µL

opentrons_96_aluminumblock_nest_wellplate_100uL

This command loads the same physical adapter and labware as the example in the Standalone Adapters section above, but it is also compatible with earlier API versions:

```
temp_combo = temp_mod.load_labware(
    "opentrons_96_aluminumblock_nest_wellplate_100uL"
)
```

New in version 2.0.

Temperature Control

The primary function of the module is to control the temperature of its deck, using set_temperature(), which takes one parameter: celsius. For example, to set the Temperature Module to 4 °C:

```
temp_mod.set_temperature(celsius=4)
```

When using set_temperature(), your protocol will wait until the target temperature is reached before proceeding to further commands. In other words, you can pipette to or from the Temperature Module when it is holding at a temperature or idle, but not while it is actively changing temperature. Whenever the module reaches its target temperature, it will hold the temperature until you set a different target or call deactivate(), which will stop heating or cooling and will turn off the fan.

Note

Your robot will not automatically deactivate the Temperature Module at the end of a protocol. If you need to deactivate the module after a protocol is completed or canceled, use the Temperature Module controls on the device detail page in the Opentrons App or run deactivate() in Jupyter notebook.

New in version 2.0.

Temperature Status

If you need to confirm in software whether the Temperature Module is holding at a temperature or is idle, use the status property:

```
temp_mod.set_temperature(celsius=90)
temp_mod.status  # "holding at target"
temp_mod.deactivate()
```

```
temp_mod.status  # "idle"
```
If you don’t need to use the status value in your code, and you have physical access to the module, you can read its status and temperature from the LED and display on the module.

Thermocycler

The Thermocycler Module provides on-deck, fully automated thermocycling, and can heat and cool very quickly during operation. The module’s block can reach and maintain temperatures between 4 and 99 °C. The module’s lid can heat up to 110 °C.

The Thermocycler is represented in code by a ThermocyclerContext object, which has methods for controlling the lid, controlling the block, and setting profiles — timed heating and cooling routines that can be repeated automatically.

The examples in this section will use a Thermocycler Module GEN2 loaded as follows:

```
tc_mod = protocol.load_module(module_name="thermocyclerModuleV2")
plate = tc_mod.load_labware(name="nest_96_wellplate_100ul_pcr_full_skirt")
```
New in version 2.13.

Lid Control

The Thermocycler can control the position and temperature of its lid.

To change the lid position, use open_lid() and close_lid(). When the lid is open, the pipettes can access the loaded labware.

You can also control the temperature of the lid. Acceptable target temperatures are between 37 and 110 °C. Use set_lid_temperature(), which takes one parameter: the target temperature (in degrees Celsius) as an integer. For example, to set the lid to 50 °C:

```
tc_mod.set_lid_temperature(temperature=50)
```
The protocol will only proceed once the lid temperature reaches 50 °C. This is the case whether the previous temperature was lower than 50 °C (in which case the lid will actively heat) or higher than 50 °C (in which case the lid will passively cool).

You can turn off the lid heater at any time with deactivate_lid().

Note

Lid temperature is not affected by Thermocycler profiles. Therefore you should set an appropriate lid temperature to hold during your profile before executing it. See Thermocycler Profiles for more information on defining and executing profiles.

New in version 2.0.

Block Control

The Thermocycler can control its block temperature, including holding at a temperature and adjusting for the volume of liquid held in its loaded plate.

Temperature

To set the block temperature inside the Thermocycler, use set_block_temperature(). At minimum you have to specify a temperature in degrees Celsius:

```
tc_mod.set_block_temperature(temperature=4)
```

If you don't specify any other parameters, the Thermocycler will hold this temperature until a new temperature is set, deactivate_block() is called, or the module is powered off.

New in version 2.0.

Hold Time

You can optionally instruct the Thermocycler to hold its block temperature for a specific amount of time. You can specify hold_time_minutes, hold_time_seconds, or both (in which case they will be added together). For example, this will set the block to 4 °C for 4 minutes and 15 seconds:

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

Note

Your protocol will not proceed to further commands while holding at a temperature. If you don't specify a hold time, the protocol will proceed as soon as the target temperature is reached.

New in version 2.0.

Block Max Volume

The Thermocycler's block temperature controller varies its behavior based on the amount of liquid in the wells of its labware. Accurately specifying the liquid volume allows the Thermocycler to more precisely control the temperature of the samples. You should set the block_max_volume parameter to the amount of liquid in the fullest well, measured in µL. If not specified, the Thermocycler will assume samples of 25 µL.

It is especially important to specify block_max_volume when holding at a temperature. For example, say you want to hold larger samples at a temperature for a short time:

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

If the Thermocycler assumes these samples are 25 µL, it may not cool them to 4 °C before starting the 20-second timer. In fact, with such a short hold time they may not reach 4 °C at all!

New in version 2.0.

Thermocycler Profiles

In addition to executing individual temperature commands, the Thermocycler can automatically cycle through a sequence of block temperatures to perform heat-sensitive reactions. These sequences are called profiles, which are defined in the Protocol API as lists of dictionaries. Each dictionary within the profile should have a temperature key, which specifies the temperature of the step, and either or both of hold_time_seconds and hold_time_minutes, which specify the duration of the step.

For example, this profile commands the Thermocycler to reach 10 °C and hold for 30 seconds, and then to reach 60 °C and hold for 45 seconds:

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

Once you have written the steps of your profile, execute it with execute_profile(). This function executes your profile steps multiple times depending on the repetitions parameter. It also takes a block_max_volume parameter, which is the same as that of the set_block_temperature() function.

For instance, a PCR prep protocol might define and execute a profile like this:

```
profile = [
    {"temperature":95, "hold_time_seconds":30},
    {"temperature":57, "hold_time_seconds":30},
    {"temperature":72, "hold_time_seconds":60}
]
tc_mod.execute_profile(steps=profile, repetitions=20, block_max_volume=32)
```

In terms of the actions that the Thermocycler performs, this would be equivalent to nesting set_block_temperature commands in a for loop:

```
for i in range(20):
    tc_mod.set_block_temperature(95, hold_time_seconds=30, block_max_volume=32)
    tc_mod.set_block_temperature(57, hold_time_seconds=30, block_max_volume=32)
    tc_mod.set_block_temperature(72, hold_time_seconds=60, block_max_volume=32)
```

However, this code would generate 60 lines in the protocol's run log, while executing a profile is summarized in a single line. Additionally, you can set a profile once and execute it multiple times (with different numbers of repetitions and maximum volumes, if needed).

Note

Temperature profiles only control the temperature of the block in the Thermocycler. You should set a lid temperature before executing the profile using set_lid_temperature().

Deck slots

Deck slots are where you place hardware items on the deck surface of your Opentrons robot. In the API, you load the corresponding items into your protocol with methods like ProtocolContext.load_labware, ProtocolContext.load_module, or ProtocolContext.load_trash_bin. When you call these methods, you need to specify which slot to load the item in.

Physical Deck Labels

Flex uses a coordinate labeling system for slots A1 (back left) through D4 (front right). Columns 1 through 3 are in the working area and are accessible by pipettes and the gripper. Column 4 is in the staging area and is only accessible by the gripper. For more information on staging area slots, see Deck Configuration below.

_images/flex-deck.svg
OT-2 uses a numeric labeling system for slots 1 (front left) through 11 (back center). The back right slot is occupied by the fixed trash.

_images/OT-2-deck.svg
API Deck Labels
The API accepts values that correspond to the physical deck slot labels on a Flex or OT-2 robot. Specify a slot in either format:

A coordinate like "A1". This format must be a string.

A number like "10" or 10. This format can be a string or an integer.

As of API version 2.15, the Flex and OT-2 formats are interchangeable. You can use either format, regardless of which robot your protocol is for. You could even mix and match formats within a protocol, although this is not recommended.

For example, these two load_labware() commands are equivalent:

protocol.load_labware("nest_96_wellplate_200ul_flat", "A1")
New in version 2.15.

protocol.load_labware("nest_96_wellplate_200ul_flat", 10)
New in version 2.0.

Both of these commands would require you to load the well plate in the back left slot of the robot.

The correspondence between deck labels is based on the relative locations of the slots. The full list of slot equivalencies is as follows:

Flex

A1

A2

A3

B1

B2

B3

C1

C2

C3

D1

D2

D3

OT-2

10

11

Trash

7

8

9

4

5

6

1

2

3

Slots A4, B4, C4, and D4 on Flex have no equivalent on OT-2.

Deck Configuration
A Flex running robot system version 7.1.0 or higher lets you specify its deck configuration on the touchscreen or in the Opentrons App. This tells the robot the positions of unpowered deck fixtures: items that replace standard deck slots. The following table lists currently supported deck fixtures and their allowed deck locations.

Fixture

Slots

Staging area slots

A3–D3

Trash bin

A1–D1, A3-D3

Waste chute

D3

Which fixtures you need to configure depend on both load methods and the effects of other methods called in your protocol. The following sections explain how to configure each type of fixture.

Staging Area Slots

Slots A4 through D4 are the staging area slots. Pipettes can't reach the staging area, but these slots are always available in the API for loading and moving labware. Using a slot in column 4 as the location argument of load_labware() or the new_location argument of move_labware() will require the corresponding staging area slot in the robot's deck configuration:

```
plate_1 = protocol.load_labware(
    load_name="corning_96_wellplate_360ul_flat", location="C3"
)  # no staging slots required
plate_2 = protocol.load_labware(
    load_name="corning_96_wellplate_360ul_flat", location="D4"
)  # one staging slot required
protocol.move_labware(
    labware=plate_1, new_location="C4"
)  # two staging slots required
```

New in version 2.16.

Since staging area slots also include a standard deck slot in column 3, they are physically incompatible with powered modules in the same row of column 3. For example, if you try to load a module in C3 and labware in C4, the API will raise an error:

```
temp_mod = protocol.load_module(
    module_name="temperature module gen2",
    location="C3"
)
staging_plate = protocol.load_labware(
    load_name="corning_96_wellplate_360ul_flat", location="C4"
)  # deck conflict error
```

It is possible to use slot D4 along with the waste chute. See the Waste Chute section below for details.

Trash Bin

In version 2.15 of the API, Flex can only have a single trash bin in slot A3. You do not have to (and cannot) load the trash in version 2.15 protocols.

Starting in API version 2.16, you must load trash bin fixtures in your protocol in order to use them. Use load_trash_bin() to load a movable trash bin. This example loads a single bin in the default location:

```
default_trash = protocol.load_trash_bin(location = "A3")
```

New in version 2.16.

Note

The TrashBin class doesn't have any callable methods, so you don't have to save the result of load_trash_bin() to a variable, especially if your protocol only loads a single trash container. Being able to reference the trash bin by name is useful when dealing with multiple trash containers.

Call load_trash_bin() multiple times to add more than one bin. See Adding Trash Containers for more information on using pipettes with multiple trash bins.

Pipettes

Opentrons pipettes are configurable devices used to move liquids throughout the working area during the execution of protocols. Flex and OT-2 each have their own pipettes, which are available for use in the Python API.

Loading 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.

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.

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 PipettesOT-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 Single-Channel GEN2

20-300

p300_single_gen2

P300 Multi-Channel GEN2

p300_multi_gen2

P1000 Single-Channel GEN2

100-1000

p1000_single_gen2

See the OT-2 Pipette Generations section 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-channel model.

Loading OT-2 Pipettes¶
This code sample loads a P1000 Single-Channel GEN2 pipette in the left mount and a P300 Single-Channel GEN2 pipette in the right mount. Each pipette uses its own 1000 µL tip rack.

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}

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.

Adding Tip Racks

The load_instrument() method includes the optional argument tip_racks. This parameter accepts a list of tip rack labware objects, which lets you to specify as many tip racks as you want. You can also edit a pipette's tip racks after loading it by setting its InstrumentContext.tip_racks property.

Note

Some methods, like configure_nozzle_layout(), reset a pipette's tip racks. See Partial Tip Pickup for more information.

The advantage of using tip_racks is twofold. First, associating tip racks with your pipette allows for automatic tip tracking throughout your protocol. Second, it removes the need to 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()
```
New in version 2.0.

See also Building Block Commands and Complex Commands.

Adding Trash Containers

The API automatically assigns a trash_container to pipettes, if one is available in your protocol. The trash_container is where the pipette will dispose tips when you call drop_tip() with no arguments. You can change the trash container, if you don't want to use the default.

One example of when you might want to change the trash container is a Flex protocol that goes through a lot of tips. In a case where the protocol uses two pipettes, you could load two trash bins and assign one to each pipette:

```
left_pipette = protocol.load_instrument(
    instrument_name="flex_8channel_1000", mount="left"
)
right_pipette = protocol.load_instrument(
    instrument_name="flex_8channel_50", mount="right"
)
left_trash = load_trash_bin("A3")
right_trash = load_trash_bin("B3")
left_pipette.trash_container = left_trash
right_pipette.trash_container = right_trash
```
Another example is a Flex protocol that uses a waste chute. Say you want to only dispose labware in the chute, and you want the pipette to drop tips in a trash bin. You can implicitly get the trash bin to be the pipette's trash_container based on load order, or you can ensure it by setting it after all the load commands:

```
pipette = protocol.load_instrument(
    instrument_name="flex_1channel_1000",
    mount="left"
)
chute = protocol.load_waste_chute()  # default because loaded first
trash = protocol.load_trash_bin("A3")
pipette.trash_container = trash  # overrides default
```

Pipette Characteristics

Each Opentrons pipette has different capabilities, which you'll want to take advantage of in your protocols. This page covers some fundamental pipette characteristics.

Multi-Channel Movement gives examples of how multi-channel pipettes move around the deck by using just one of their channels as a reference point. Taking this into account is important for commanding your pipettes to perform actions in the correct locations.

Pipette Flow Rates discusses how quickly each type of pipette can handle liquids. The defaults are designed to operate quickly, based on the pipette's hardware and assuming that you're handling aqueous liquids. You can speed up or slow down a pipette's flow rate to suit your protocol's needs.

Finally, the volume ranges of pipettes affect what you can do with them. The volume ranges for current pipettes are listed on the Loading Pipettes page. The OT-2 Pipette Generations section of this page describes how the API behaves when running protocols that specify older OT-2 pipettes.

Multi-Channel Movement

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

To keep the protocol API consistent when using single- and multi-channel pipettes, commands treat the back left channel of a multi-channel pipette as its primary channel. Location arguments of pipetting commands use the primary channel. The InstrumentContext.configure_nozzle_layout() method can change the pipette's primary channel, using its start parameter. See Partial Tip Pickup for more information.

With a pipette's default settings, you can generally access the wells indicated in the table below. Moving to any other well may cause the pipette to crash.

Channels

96-well plate

384-well plate

1

Any well, A1–H12

Any well, A1–P24

8

A1–A12

A1–B24

96

A1 only

A1–B2

Also, you should apply any location offset, such as Well.top() or Well.bottom(), to the well accessed by the primary channel. Since all of the pipette's channels move together, each channel will have the same offset relative to the well that it is over.

Finally, because each multi-channel pipette has only one motor, they 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":"2.17"}

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()
```

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 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.

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.

Note
In API version 2.13 and earlier, InstrumentContext.speed offered similar functionality to .flow_rate. It attempted to 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.

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).

Building Block Commands

Building block commands execute some of the most basic actions that your robot can complete. But basic doesn't mean these commands lack capabilities. They perform important tasks in your protocols. They're also foundational to the complex commands that help you combine multiple actions into fewer lines of code.

Manipulating Pipette Tips

Your robot needs to attach a disposable tip to the pipette before it can aspirate or dispense liquids. The API provides three basic functions that help the robot attach and manage pipette tips during a protocol run. These methods are InstrumentContext.pick_up_tip(), InstrumentContext.drop_tip(), and InstrumentContext.return_tip(). Respectively, these methods tell the robot to pick up a tip from a tip rack, drop a tip into the trash (or another location), and return a tip to its location in the tip rack.

The following sections demonstrate how to use each method and include sample code. The examples used here assume that you've loaded the pipettes and labware from the basic protocol template.

Picking Up a Tip

To pick up a tip, call the pick_up_tip() method without any arguments:

```
pipette.pick_up_tip()
```

This simple statement works because the variable tiprack_1 in the sample protocol includes the on-deck location of the tip rack (OT-2 location=3) and the pipette variable includes the argument tip_racks=[tiprack_1]. Given this information, the robot moves to the tip rack and picks up a tip from position A1 in the rack. On subsequent calls to pick_up_tip(), the robot will use the next available tip. For example:

```
pipette.pick_up_tip()  # picks up tip from rack location A1
pipette.drop_tip()     # drops tip in trash bin
pipette.pick_up_tip()  # picks up tip from rack location B1
pipette.drop_tip()     # drops tip in trash bin
```

If you omit the tip_rack argument from the pipette variable, the API will raise an error. You must pass in the tip rack's location to pick_up_tip like this:

```
pipette.pick_up_tip(tiprack_1["A1"])
pipette.drop_tip()
pipette.pick_up_tip(tiprack_1["B1"])
```

If coding the location of each tip seems inefficient or tedious, try using a for loop to automate a sequential tip pick up process. When using a loop, the API keeps track of tips and manages tip pickup for you. But pick_up_tip is still a powerful feature. It gives you direct control over tip use when that's important in your protocol.

Automating Tip Pick Up

When used with Python's range class, a for loop brings automation to the tip pickup and tracking process. It also eliminates the need to call pick_up_tip() multiple times. For example, this snippet tells the robot to sequentially use all the tips in a 96-tip rack:

```
for i in range(96):
    pipette.pick_up_tip()
    # liquid handling commands
    pipette.drop_tip()
```

If your protocol requires a lot of tips, add a second tip rack to the protocol. Then, associate it with your pipette and increase the number of repetitions in the loop. The robot will work through both racks.

Dropping a Tip

To drop a tip in the pipette's trash container, call the drop_tip() method with no arguments:

```
pipette.pick_up_tip()
```
You can also specify where to drop the tip by passing in a location. For example, this code drops a tip in the trash bin and returns another tip to to a previously used well in a tip rack:

```
pipette.pick_up_tip()          # picks up tip from rack location A1
pipette.drop_tip()             # drops tip in trash bin
pipette.pick_up_tip()          # picks up tip from rack location B1
pipette.drop_tip(tiprack["A1"])  # drops tip in rack location A1
```

Returning a Tip

To return a tip to its original location, call the return_tip() method with no arguments:

```
pipette.return_tip()
```
New in version 2.0.

Note

You can't return tips with a pipette that's configured to use partial tip pickup. This restriction ensures that the pipette has clear access to unused tips. For example, a 96-channel pipette in column configuration can't reach column 2 unless column 1 is empty.

If you call return_tip() while using partial tip pickup, the API will raise an error. Use drop_tip() to dispose the tips instead.

Working With Used Tips

Currently, the API considers tips as "used" after being picked up. For example, if the robot picked up a tip from rack location A1 and then returned it to the same location, it will not attempt to pick up this tip again, unless explicitly specified. Instead, the robot will pick up a tip starting from rack location B1. For example:

```
pipette.pick_up_tip()              # picks up tip from rack location A1
pipette.return_tip()               # drops tip in rack location A1
pipette.pick_up_tip()              # picks up tip from rack location B1
pipette.drop_tip()                 # drops tip in trash bin
pipette.pick_up_tip(tiprack_1["A1"]) # picks up tip from rack location A1
```
Early API versions treated returned tips as unused items. They could be picked up again without an explicit argument. For example:

```
pipette.pick_up_tip()  # picks up tip from rack location A1
pipette.return_tip()   # drops tip in rack location A1
pipette.pick_up_tip()  # picks up tip from rack location A1
```

Liquid control

After attaching a tip, your robot is ready to aspirate, dispense, and perform other liquid handling tasks. The API includes methods that help you perform these actions and the following sections show how to use them. The examples used here assume that you've loaded the pipettes and labware from the basic protocol template.

Aspirate

To draw liquid up into a pipette tip, call the InstrumentContext.aspirate() method. Using this method, you can specify the aspiration volume in µL, the well location, and pipette flow rate. Other parameters let you position the pipette within a well. For example, this snippet tells the robot to aspirate 200 µL from well location A1.

```
pipette.pick_up_tip()
pipette.aspirate(200, plate["A1"])
```

If the pipette doesn't move, you can specify an additional aspiration action without including a location. To demonstrate, this code snippet pauses the protocol, automatically resumes it, and aspirates a second time from plate["A1"]).

```
pipette.pick_up_tip()
pipette.aspirate(200, plate["A1"])
protocol.delay(seconds=5) # pause for 5 seconds
pipette.aspirate(100)     # aspirate 100 µL at current position
```

Now our pipette holds 300 µL.

Aspirate by Well or Location

The aspirate() method includes a location parameter that accepts either a Well or a Location.

If you specify a well, like plate["A1"], the pipette will aspirate from a default position 1 mm above the bottom center of that well. To change the default clearance, first set the aspirate attribute of well_bottom_clearance:

```
pipette.pick_up_tip
pipette.well_bottom_clearance.aspirate = 2  # tip is 2 mm above well bottom
pipette.aspirate(200, plate["A1"])
```

You can also aspirate from a location along the center vertical axis within a well using the Well.top() and Well.bottom() methods. These methods move the pipette to a specified distance relative to the top or bottom center of a well:

```
pipette.pick_up_tip()
depth = plate["A1"].bottom(z=2) # tip is 2 mm above well bottom
pipette.aspirate(200, depth)
```

Aspiration Flow Rates

Flex and OT-2 pipettes aspirate at default flow rates measured in µL/s. Specifying the rate parameter multiplies the flow rate by that value. As a best practice, don't set the flow rate higher than 3x the default. For example, this code causes the pipette to aspirate at twice its normal rate:

```
pipette.aspirate(200, plate["A1"], rate=2.0)
```

Dispense

To dispense liquid from a pipette tip, call the InstrumentContext.dispense() method. Using this method, you can specify the dispense volume in µL, the well location, and pipette flow rate. Other parameters let you position the pipette within a well. For example, this snippet tells the robot to dispense 200 µL into well location B1.

```
pipette.dispense(200, plate["B1"])
```

Note

In API version 2.16 and earlier, you could pass a volume argument to dispense() greater than what was aspirated into the pipette. In this case, the API would ignore volume and dispense the pipette's current_volume. The robot would not move the plunger lower as a result.

In version 2.17 and later, passing such values raises an error.

To move the plunger a small extra amount, add a push out. Or to move it a large amount, use blow out.

If the pipette doesn't move, you can specify an additional dispense action without including a location. To demonstrate, this code snippet pauses the protocol, automatically resumes it, and dispense a second time from location B1.

```
pipette.dispense(100, plate["B1"])
protocol.delay(seconds=5) # pause for 5 seconds
pipette.dispense(100)     # dispense 100 µL at current position
```

Dispense by Well or Location

The dispense() method includes a location parameter that accepts either a Well or a Location.

If you specify a well, like plate["B1"], the pipette will dispense from a default position 1 mm above the bottom center of that well. To change the default clearance, you would call well_bottom_clearance:

```
pipette.well_bottom_clearance.dispense=2 # tip is 2 mm above well bottom
pipette.dispense(200, plate["B1"])
```

You can also dispense from a location along the center vertical axis within a well using the Well.top() and Well.bottom() methods. These methods move the pipette to a specified distance relative to the top or bottom center of a well:

```
depth = plate["B1"].bottom(z=2) # tip is 2 mm above well bottom
pipette.dispense(200, depth)
```

Dispense Flow Rates

Flex and OT-2 pipettes dispense at default flow rates measured in µL/s. Adding a number to the rate parameter multiplies the flow rate by that value. As a best practice, don't set the flow rate higher than 3x the default. For example, this code causes the pipette to dispense at twice its normal rate:

```
pipette.dispense(200, plate["B1"], rate=2.0)
```

Push Out After Dispense

The optional push_out parameter of dispense() helps ensure all liquid leaves the tip. Use push_out for applications that require moving the pipette plunger lower than the default, without performing a full blow out.

For example, this dispense action moves the plunger the equivalent of an additional 5 µL beyond where it would stop if push_out was set to zero or omitted:

```
pipette.pick_up_tip()
pipette.aspirate(100, plate["A1"])
pipette.dispense(100, plate["B1"], push_out=5)
pipette.drop_tip()
```

Blow Out

To blow an extra amount of air through the pipette's tip, call the InstrumentContext.blow_out() method. You can use a specific well in a well plate or reservoir as the blowout location. If no location is specified, the pipette will blowout from its current well position:

```
pipette.blow_out()
```

You can also specify a particular well as the blowout location:

```
pipette.blow_out(plate["B1"])
```

Many protocols use a trash container for blowing out the pipette. You can specify the pipette's current trash container as the blowout location by using the InstrumentContext.trash_container property:

```
pipette.blow_out(pipette.trash_container)
```

Touch Tip

The InstrumentContext.touch_tip() method moves the pipette so the tip touches each wall of a well. A touch tip procedure helps knock off any droplets that might cling to the pipette's tip. This method includes optional arguments that allow you to control where the tip will touch the inner

walls of a well and the touch speed. Calling touch_tip() without arguments causes the pipette to touch the well walls from its current location:

```
pipette.touch_tip()
```

Touch Location
These optional location arguments give you control over where the tip will touch the side of a well.

This example demonstrates touching the tip in a specific well:

```
pipette.touch_tip(plate["B1"])
```
This example uses an offset to set the touch tip location 2mm below the top of the current well:

```
pipette.touch_tip(v_offset=-2)
```
This example moves the pipette 75% of well's total radius and 2 mm below the top of well:

```
pipette.touch_tip(plate["B1"],
                  radius=0.75,
                  v_offset=-2)
```
The touch_tip feature allows the pipette to touch the edges of a well gently instead of crashing into them. It includes the radius argument. When radius=1 the robot moves the centerline of the pipette's plunger axis to the edge of a well. This means a pipette tip may sometimes touch the well wall too early, causing it to bend inwards. A smaller radius helps avoid premature wall collisions and a lower speed produces gentler motion. Different liquid droplets behave differently, so test out these parameters in a single well before performing a full protocol run.

Warning
Do not set the radius value greater than 1.0. When radius is > 1.0, the robot will forcibly move the pipette tip across a well wall or edge. This type of aggressive movement can damage the pipette tip and the pipette.

Touch Speed
Touch speed controls how fast the pipette moves in mm/s during a touch tip step. The default movement speed is 60 mm/s, the minimum is 1 mm/s, and the maximum is 80 mm/s. Calling touch_tip without any arguments moves a tip at the default speed in the current well:

```
pipette.touch_tip()
```
This example specifies a well location and sets the speed to 20 mm/s:

```
pipette.touch_tip(plate["B1"], speed=20)
```
This example uses the current well and sets the speed to 80 mm/s:

```
pipette.touch_tip(speed=80)
```

Mix

The mix() method aspirates and dispenses repeatedly in a single location. It's designed to mix the contents of a well together using a single command rather than using multiple aspirate() and dispense() calls. This method includes arguments that let you specify the number of times to mix, the volume (in µL) of liquid, and the well that contains the liquid you want to mix.

This example draws 100 µL from the current well and mixes it three times:

```
pipette.mix(repetitions=3, volume=100)
```

This example draws 100 µL from well B1 and mixes it three times:

```
pipette.mix(3, 100, plate["B1"])
```

This example draws an amount equal to the pipette's maximum rated volume and mixes it three times:

```
pipette.mix(repetitions=3)
```

Air Gap

The InstrumentContext.air_gap() method tells the pipette to draw in air before or after a liquid. Creating an air gap helps keep liquids from seeping out of a pipette after drawing it from a well. This method includes arguments that give you control over the amount of air to aspirate and the pipette's height (in mm) above the well. By default, the pipette moves 5 mm above a well before aspirating air. Calling air_gap() with no arguments uses the entire remaining volume in the pipette.

This example aspirates 200 µL of air 5 mm above the current well:

```
pipette.air_gap(volume=200)
```

This example aspirates 200 µL of air 20 mm above the the current well:

```
pipette.air_gap(volume=200, height=20)
```

This example aspirates enough air to fill the remaining volume in a pipette:

```
pipette.air_gap()
```

Utility Commands

With utility commands, you can control various robot functions such as pausing or delaying a protocol, checking the robot's door, turning robot lights on/off, and more. The following sections show you how to these utility commands and include sample code. The examples used here assume that you've loaded the pipettes and labware from the basic protocol template.

Delay and Resume

Call the ProtocolContext.delay() method to insert a timed delay into your protocol. This method accepts time increments in seconds, minutes, or combinations of both. Your protocol resumes automatically after the specified time expires.

This example delays a protocol for 10 seconds:

protocol.delay(seconds=10)
This example delays a protocol for 5 minutes:

protocol.delay(minutes=5)
This example delays a protocol for 5 minutes and 10 seconds:

protocol.delay(minutes=5, seconds=10)
Pause Until Resumed
Call the ProtocolContext.pause() method to stop a protocol at a specific step. Unlike a delay, pause() does not restart your protocol automatically. To resume, you'll respond to a prompt on the touchscreen or in the Opentrons App. This method also lets you specify an optional message that provides on-screen or in-app instructions on how to proceed. This example inserts a pause and includes a brief message:

protocol.pause("Remember to get more pipette tips")
New in version 2.0.

Homing
Homing commands the robot to move the gantry, a pipette, or a pipette plunger to a defined position. For example, homing the gantry moves it to the back right of the working area. With the available homing methods you can home the gantry, home the mounted pipette and plunger, and home the pipette plunger. These functions take no arguments.

To home the gantry, call ProtocolContext.home():

protocol.home()
To home a specific pipette's Z axis and plunger, call InstrumentContext.home():

pipette = protocol.load_instrument("flex_1channel_1000", "right")
pipette.home()
To home a specific pipette's plunger only, you can call InstrumentContext.home_plunger():

pipette = protocol.load_instrument("flex_1channel_1000", "right")
pipette.home_plunger()
New in version 2.0.

Comment

Call the ProtocolContext.comment() method if you want to write and display a brief message in the Opentrons App during a protocol run:

```
protocol.comment("Hello, world!")
```

Complex commands

Complex liquid handling commands combine multiple building block commands into a single method call. These commands make it easier to handle larger groups of wells and repeat actions without having to write your own control flow code. They integrate tip-handling behavior and can pick up, use, and drop multiple tips depending on how you want to handle your liquids. They can optionally perform other actions, like adding air gaps, knocking droplets off the tip, mixing, and blowing out excess liquid from the tip.

There are three complex liquid handling commands, each optimized for a different liquid handling scenario:

InstrumentContext.transfer()

InstrumentContext.distribute()

InstrumentContext.consolidate()

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 around such situations by making multiple calls to transfer() in sequence or by using a list of volumes to skip certain wells.

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:

Pick up a tip.

Aspirate from the first source well.

Dispense in the first destination well.

Repeat the pattern of aspirating and dispensing, as needed.

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:

Pick up a tip.

Aspirate enough to dispense in all the destination wells. This aspirate includes a disposal volume.

Dispense in the first destination well.

Continue to dispense in destination wells.

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.

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

Pick up a tip.

Aspirate from the first source well.

Continue aspirating from source wells.

Dispense in the destination well.

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.
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],
    dest=plate.rows()[1],
)
```

Source

A1

A2

A3

A4

A5

A6

A7

A8

A9

A10

A11

A12

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

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 by repeating transfers in a for loop:

```
for i in range(12):
   pipette.transfer(
      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).

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:

Pick up tip

Mix at source

Aspirate from source

Touch tip at source

Air gap

Dispense into destination

Mix at destination

Touch tip at destination

Blow out

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"]],
    touch_tip=True,
)
```

Compared to the list of all possible actions, this code will only perform the following:

Pick up tip

Aspirate from source

Touch tip at source

Dispense into destination

Touch tip at destination

Blow out

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!

Remember that distribute() includes a disposal volume by default, and this can affect the number of times the pipette refills its tip. Say you want to distribute 80 µL to each of the 12 wells in row A of a plate. That's 960 µL total — less than the capacity of the pipette — but the 100 µL disposal volume will cause the pipette to refill.

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.

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

One reason to set new_tip="always" is to avoid cross-contamination between wells. However, you should always do a dry run of your protocol to test that the pipette is picking up and dropping tips in the way that your application requires.

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.

By default, disposal_volume is the minimum volume of the pipette, but you can set it to any amount:

```
pipette.distribute(
    volume=100,
    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.

Air-gapping behavior is different for each complex command. The different behaviors all serve the same purpose, which is to never leave the pipette holding liquid at the very bottom of the tip. This helps keep liquids from seeping out of the pipette.

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
Dispensing 170.0 uL into B1 of well plate on 2 at 92.86 uL/sec
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

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:

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

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 contact the liquid. Similar to the 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.

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

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

All positions relative to labware are adjusted automatically based on labware offset data. Calculate labware offsets by running Labware Position Check during protocol setup, either in the Opentrons App or on the Flex touchscreen. Version 6.0.0 and later of the robot software can apply previously calculated offsets on the same robot for the same labware type and 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.

Small, direct movements can be useful for working inside of a well, without having the tip exit and re-enter the well. This code sample demonstrates how to move the pipette to a well, make direct movements inside that well, and then move on to a different well:

```
pipette.move_to(plate["A1"].top())
pipette.move_to(plate["A1"].bottom(1), force_direct=True)
pipette.move_to(plate["A1"].top(-2), force_direct=True)
pipette.move_to(plate["A2"].top())
```
New in version 2.0.

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 case, you can reduce the gantry speed for a specific pipette by setting InstrumentContext.default_speed like this:

```
pipette.move_to(plate["A1"].top())  # move to the first well at default speed
pipette.default_speed = 100         # reduce pipette speed
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.

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("2.17")
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("2.17")
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:

Create a "dummy" protocol that loads your labware and has each used pipette pick up a tip from a tip rack.

Import the dummy protocol to the Opentrons App.

Run Labware Position Check from the app or touchscreen.

Add the offsets to your code with set_offset().

Creating the dummy protocol requires you to:

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().

Define a run() function.

Load all of your labware in their initial locations.

Load your smallest capacity pipette and specify its tip_racks.

Call pick_up_tip(). Labware Position Check can't run if you don't pick up a tip.

For example, the following dummy protocol will use a P300 Single-Channel GEN2 pipette to enable Labware Position Check for an OT-2 tip rack, NEST reservoir, and NEST flat well plate.

```
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.

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
This code only loads the instruments and labware listed above, and performs no other actions. Many code snippets from elsewhere in the documentation will run without modification when added at the bottom of this template. You can also use it to start writing and testing your own code.

FlexOT-2
```
from opentrons import protocol_api
```

```
metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    # load tip rack in deck slot 3
    tiprack = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul", location=3
    )
    # attach pipette to left mount
    pipette = protocol.load_instrument(
        instrument_name="p300_single_gen2",
        mount="left",
        tip_racks=[tiprack]
    )
    # load well plate in deck slot 2
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat", location=2
    )
    # load reservoir in deck slot 1
    reservoir = protocol.load_labware(
        load_name="usascientific_12_reservoir_22ml", location=1
    )
    # 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.

FlexOT-2

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
            load_name="opentrons_96_tiprack_300ul",
            location=2)
    p300 = protocol.load_instrument(
            instrument_name="p300_single",
```

```
        mount="left",
        tip_racks=[tiprack_1])

    p300.pick_up_tip()
    p300.aspirate(100, plate["A1"])
    p300.dispense(100, plate["B1"])
    p300.drop_tip()
```

Advanced Method

This protocol accomplishes the same thing as the previous example, but does it a little more efficiently. Notice how it uses 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.

FlexOT-2

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
            load_name="opentrons_96_tiprack_300ul",
            location=2)
    p300 = protocol.load_instrument(
        instrument_name="p300_single",
        mount="left",
        tip_racks=[tiprack_1])
    # transfer 100 µL from well A1 to well B1
    p300.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.

FlexOT-2

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}
```

```
def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
        location=2)
    reservoir = protocol.load_labware(
        load_name="usascientific_12_reservoir_22ml",
        location=4)
    p300 = protocol.load_instrument(
        instrument_name="p300_single",
        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):
        p300.distribute(200, reservoir.wells()[i], plate.rows()[i])
```

Notice here how Python's range class (e.g., range(8)) determines how many times the code loops. 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 in the row, which is H1=7.

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.

FlexOT-2

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
        location=2)
```

```
    reservoir = protocol.load_labware(
        load_name="usascientific_12_reservoir_22ml",
        location=3)
    p300 = protocol.load_instrument(
        instrument_name="p300_single",
        mount="right",
        tip_racks=[tiprack_1])

    p300.pick_up_tip()

    # aspirate from the first 5 wells
    for well in reservoir.wells()[:5]:
        p300.aspirate(volume=35, location=well)
        p300.air_gap(10)

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

    p300.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.

FlexOT-2

```
from opentrons import protocol_api

metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
        location=2)
    tiprack_2 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
        location=3)
    reservoir = protocol.load_labware(
```

```
        load_name="usascientific_12_reservoir_22ml",
        location=4)
    p300 = protocol.load_instrument(
        instrument_name="p300_single",
        mount="right",
        tip_racks=[tiprack_1, tiprack_2])
    # Dispense diluent
    p300.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]
        source = reservoir.wells()[i]
        row = plate.rows()[i]

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

    # dilute the sample down the column
    p300.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.

FlexOT-2

```
from opentrons import protocol_api
metadata = {"apiLevel": "2.17"}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware(
        load_name="corning_96_wellplate_360ul_flat",
        location=1)
    tiprack_1 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
        location=2)
    tiprack_2 = protocol.load_labware(
        load_name="opentrons_96_tiprack_300ul",
```

```
    location=3)
  reservoir = protocol.load_labware(
    load_name="usascientific_12_reservoir_22ml",
    location=4)
  p300 = protocol.load_instrument(
    instrument_name="p300_single",
    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,
    89, 90, 91, 92, 93, 94, 95, 96
    ]

  p300.distribute(water_volumes, reservoir["A12"], plate.wells())
```

Transferring liquid basics

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

```
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": "2.17"}

# protocol run function
```

```
def run(protocol: protocol_api.ProtocolContext):
    # labware
    plate = protocol.load_labware(
        "corning_96_wellplate_360ul_flat", location="1"
    )
    tiprack = protocol.load_labware(
        "opentrons_96_tiprack_300ul", location="2"
    )

    # 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:

Gives the name, contact information, and a brief description for the protocol.

Indicates the protocol should run on an OT-2 robot, using API version 2.17.

Tells the robot that there is:
A 96-well flat plate in slot 1.

A rack of 300 µL tips in slot 2.

A single-channel 300 µL pipette attached to the left mount, which should pick up tips from the aforementioned rack.

Tells the robot to act by:
Picking up the first tip from the tip rack.

Aspirating 100 µL of liquid from well A1 of the plate.

Dispensing 100 µL of liquid into well B1 of the plate.

Dropping the tip in the trash.

Serial dilution

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.

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_15ml`
Well plate	NEST 96 Well Plate 200 µL Flat	`nest_96_wellplate_200ul_flat`
Flex tip rack	Opentrons Flex Tips, 200 µL	`opentrons_flex_96_tiprack_200ul`
OT-2 tip rack	Opentrons 96 Tip Rack	`opentrons_96_tiprack_300ul`

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 2.16, so specify:

```
metadata = {"apiLevel": "2.16"}
```

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": "2.16",
   "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": "2.16"}
```

OT-2: The requirements block is optional, but including it is a recommended best practice, particularly if you're using API version 2.15 or greater. If you do use it, remember to remove the API version from the metadata. For example:

requirements = {"robotType": "OT-2", "apiLevel": "2.16"}
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.

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.

Trash Bin
Flex and OT-2 both come with a trash bin for disposing used tips.

The OT-2 trash bin is fixed in slot 12. Since it can't go anywhere else on the deck, you don't need to write any code to tell the API where it is.

Pipettes
Next you'll specify what pipette to use in the protocol. Loading a pipette is done with the load_instrument() method, which 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:

```
# 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.

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:

Measure out equal amounts of diluent from the reservoir to every well on the plate.

Measure out equal amounts of solution from the reservoir into wells in the first column of the plate.

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.

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))
```

As before, the first argument specifies to transfer 100 µL. The second argument is the source, column 2 of the reservoir. The third argument is the destination, the element at index 0 of the current row. Since Python lists are zero-indexed, but columns on labware start numbering at 1, this will be well A1 on the first time through the loop, B1 the second time, and so on. The fourth argument specifies to mix 3 times with 50 µL of fluid each time.

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.

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).