opentrons

You are currently viewing the documentation for version 2 of the Opentrons OT-2 Python Protocol API. To view documentation for version 1 of the API, click here.

API Version 2 Reference

Protocols and Instruments

class `opentrons.protocol_api.contexts.`**`ProtocolContext`**(*implementation: opentrons.protocols.context.protocol.AbstractProtocol, loop: asyncio.events.AbstractEventLoop = None, broker=None, api_version: Optional[opentrons.protocols.api_support.types.APIVersion] = None*)

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

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

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

New in version 2.0.

property **`api_version`**

Return the API version supported by this protocol context.

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

New in version 2.0.

classmethod **`build_using`**(*implementation: opentrons.protocols.context.protocol.AbstractProtocol, protocol: Union[opentrons.protocols.types.JsonProtocol, opentrons.protocols.types.PythonProtocol], *args, **kwargs*)

Build an API instance for the specified parsed protocol

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

property **`bundled_data`**

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

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

New in version 2.0.

opentrons

New in version 2.0.

commands(*self*)

New in version 2.0.

comment(*self*, *msg*)

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

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

New in version 2.0.

connect(*self*, *hardware: opentrons.hardware_control.api.API*)

Connect to a running hardware API.

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

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

New in version 2.0.

property **deck**

The object holding the deck layout of the robot.

This object behaves like a dictionary with keys for both numeric and string slot numbers (for instance, `protocol.deck[1]` and `protocol.deck['1']` will both return the object in slot 1). If nothing is loaded into a slot, `None` will be present. This object is useful for determining if a slot in the deck is free. Rather than filtering the objects in the deck map yourself, you can also use **`loaded_labwares`** to see a dict of labwares and **`loaded_modules`** to see a dict of modules. For advanced control you can delete an item of labware from the deck with e.g. `del protocol.deck['1']` to free a slot for new labware. (Note that for each slot only the last labware used in a command will be available for calibration in the OpenTrons UI, and that the tallest labware on the deck will be calculated using only currently loaded labware, meaning that the labware loaded should always reflect the labware physically on the deck (or be higher than the labware on the deck).

New in version 2.0.

delay(*self*, *seconds=0*, *minutes=0*, *msg=None*)

Delay protocol execution for a specific amount of time.

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

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

New in version 2.0.

disconnect(*self*)

Disconnect from currently-connected hardware and simulate instead

opentrons

Returns True if the robot door is closed

New in version 2.5.

property **fixed_trash**

The trash fixed to slot 12 of the robot deck.

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

New in version 2.0.

home(*self*)

Homes the robot.

New in version 2.0.

is_simulating(*self*) → bool

New in version 2.0.

load_instrument(*self, instrument_name: str, mount: Union[opentrons.types.Mount, str], tip_racks: List[opentrons.protocol_api.labware.Labware] = None, replace: bool = False*) → 'InstrumentContext'

Load a specific instrument required by the protocol.

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

Parameters:
- **instrument_name** (*str*) – The name of the instrument model, or a prefix. For instance, 'p10_single' may be used to request a P10 single regardless of the version.
- **mount** (*types.Mount or str*) – The mount in which this instrument should be attached. This can either be an instance of the enum type **types.Mount** or one of the strings *'left'* and *'right'*.
- **tip_racks** (List[**Labware**]) – A list of tip racks from which to pick tips if **InstrumentContext.pick_up_tip()** is called without arguments.
- **replace** (*bool*) – Indicate that the currently-loaded instrument in *mount* (if such an instrument exists) should be replaced by *instrument_name*.

New in version 2.0.

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

Load a labware onto the deck given its name.

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

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

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

opentrons

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

New in version 2.0.

load_labware_from_definition(*self, labware_def: 'LabwareDefinition', location: Union[int, str], label: Union[str, NoneType] = None*) → opentrons.protocol_api.labware.Labware

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

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

Parameters:
- **labware_def** – The labware definition to load
- **location** (*int or str*) – The slot into which to load the labware such as 1 or '1'
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.

New in version 2.0.

load_module(*self, module_name: str, location: Union[int, str, NoneType] = None, configuration: Union[str, NoneType] = None*) → Union[ForwardRef('TemperatureModuleContext'), ForwardRef('MagneticModuleContext'), ForwardRef('ThermocyclerContext')]

Load a module onto the deck given its name.

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

A map of deck positions to loaded modules can be accessed later using `loaded_modules`.

Parameters:
- **module_name** (*str*) – The name or model of the module.
- **location** (*str or int or None*) – The location of the module. This is usually the name or number of the slot on the deck where you will be placing the module. Some modules, like the Thermocycler, are only valid in one deck location. You do not have to specify a location when loading a Thermocycler - it will always be in Slot 7.
- **configuration** – Used to specify the slot configuration of the Thermocycler. Only Valid in Python API Version 2.4 and later. If you wish to use the non-full plate configuration, you must pass in the key word value *semi*

Returns ModuleContext: The loaded and initialized `ModuleContext`.

New in version 2.0.

property **loaded_instruments**

Get the instruments that have been loaded into the protocol.

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

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

New in version 2.0.

opentrons

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

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

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

New in version 2.0.

property **loaded_modules**

Get the modules loaded into the protocol context.

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

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

New in version 2.0.

property **max_speeds**

Per-axis speed limits when moving this instrument.

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

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

For instance,

```
def run(protocol):
    protocol.comment(str(right.max_speeds))  # '{}' - all default
    protocol.max_speeds['A'] = 10  # limit max speed of
                                   # right pipette Z to 10mm/s
    del protocol.max_speeds['A']  # reset to default
    protocol.max_speeds['X'] = 10  # limit max speed of x to
                                   # 10 mm/s
    protocol.max_speeds['X'] = None  # reset to default
```

New in version 2.0.

pause(*self*, *msg=None*)

Pause execution of the protocol until resume is called.

opentrons

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

New in version 2.0.

property **rail_lights_on**

Returns True if the rail lights are on

New in version 2.5.

resume(*self*)

Resume a previously-paused protocol

New in version 2.0.

set_rail_lights(*self*, *on: bool*)

Controls the robot rail lights

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

New in version 2.5.

temp_connect(*self, hardware: Union[opentrons.hardware_control.thread_manager.ThreadManager, opentrons.hardware_control.adapters.SynchronousAdapter]*)

Connect temporarily to the specified hardware controller.

This should be used as a context manager:

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

class opentrons.protocol_api.contexts.**InstrumentContext**(*implementation: AbstractInstrument*, *ctx: ProtocolContext*, *broker: Broker*, *log_parent: logging.Logger*, *at_version: APIVersion*, *tip_racks: List[Labware] = None*, *trash: Optional[Labware] = None*)

A context for a specific pipette or instrument.

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

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

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

New in version 2.0.

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

Pull air into the pipette current tip at the current location

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

opentrons

Returns: This instance

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

New in version 2.0.

property **api_version**

New in version 2.0.

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

Aspirate a given volume of liquid from the specified location, using this pipette.

Parameters:
- **volume** (*int or float*) – The volume to aspirate, in microliters (µL). If 0 or unspecified, defaults to the highest volume possible with this pipette and its currently attached tip.
- **location** – Where to aspirate from. If *location* is a **Well**, the robot will aspirate from **well_bottom_clearance.aspirate** mm above the bottom of the well. If *location* is a **Location** (i.e. the result of **Well.top()** or **Well.bottom()**), the robot will aspirate from the exact specified location. If unspecified, the robot will aspirate from the current position.
- **rate** (*float*) – A relative modifier for how quickly to aspirate liquid. The flow rate for this aspirate will be *rate* * **flow_rate.aspirate**. If not specified, defaults to 1.0.

Returns: This instance.

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

New in version 2.0.

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

Blow liquid out of the tip.

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

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

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

opentrons

property **channels**

The number of channels on the pipette.

New in version 2.0.

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

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

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

Returns: This instance

New in version 2.0.

property **current_volume**

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

New in version 2.0.

property **default_speed**

The speed at which the robot's gantry moves.

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

New in version 2.0.

delay(*self*)

New in version 2.0.

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

Dispense a volume of liquid (in microliters/uL) using this pipette into the specified location.

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

Parameters:
- **volume** (*int or float*) – The volume of liquid to dispense, in microliters. If not specified, defaults to **`current_volume`**.
- **location** – Where to dispense into. If *location* is a **`Well`**, the robot will dispense into **`well_bottom_clearance.dispense`** mm above the bottom of the well. If *location* is a **`Location`** (i.e. the result of **`Well.top()`** or **`Well.bottom()`**), the robot will dispense into the exact specified location. If unspecified, the robot will dispense into the current position.

opentrons

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

New in version 2.0.

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

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

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

Returns: This instance

New in version 2.0.

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

Drop the current tip.

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

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

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

Parameters:
- **location** (**`types.Location`** or **`Well`** or None) – The location to drop the tip
- **home_after** –
  Whether to home this pipette's plunger after dropping the tip. Defaults to `True`.
  Setting `home_after=False` saves waiting a couple of seconds after the pipette drops the tip, but risks causing other problems.

> **Warning:**
>
> Only set `home_after=False` if:

opentrons

- You understand the risks described below.

The ejector shroud that pops the tip off the end of the pipette is driven by the plunger's stepper motor. Sometimes, the strain of ejecting the tip can make that motor *skip* and fall out of sync with where the robot thinks it is.

Homing the plunger fixes this, so, to be safe, we normally do it after every tip drop.

If you set `home_after=False` to disable homing the plunger, and the motor happens to skip, you might see problems like these until the next time the plunger is homed:

- The run might halt with a "hard limit" error message.
- The pipette might aspirate or dispense the wrong volumes.
- The pipette might not fully drop subsequent tips.

GEN1 pipettes are especially vulnerable to this skipping, so you should never set `home_after=False` with a GEN1 pipette.

Even on GEN2 pipettes, the motor can still skip. So, always extensively test `home_after=False` with your particular pipette and your particular tips before relying on it.

Returns: This instance

New in version 2.0.

property **flow_rate**

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

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

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

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

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

New in version 2.0.

property **has_tip**

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

New in version 2.7.

Type: returns

home(*self*) → 'InstrumentContext'

Home the robot.

Returns:

opentrons

home_plunger(*self*) → 'InstrumentContext'

Home the plunger associated with this mount

Returns: This instance.

New in version 2.0.

property **hw_pipette**

View the information returned by the hardware API directly.

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

New in version 2.0.

property **max_volume**

The maximum volume, in microliters (µL), that this pipette can hold.

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

New in version 2.0.

property **min_volume**

New in version 2.0.

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

Mix a volume of liquid (uL) using this pipette, by repeatedly aspirating and dispensing in the same place.

Parameters:
- **repetitions** – how many times the pipette should mix (default: 1)
- **volume** – number of microliters to mix. If 0 or unspecified, defaults to the highest volume possible with this pipette and its currently attached tip.
- **location** (*types.Location*) – a Well or a position relative to well. e.g, *plate.rows()[0][0].bottom()*. If unspecified, the pipette will mix from its current position.
- **rate** – A relative modifier for how quickly to aspirate and dispense liquid during this mix. When aspirating, the flow rate will be *rate* * `flow_rate.aspirate`, and when dispensing, it will be *rate* * `flow_rate.dispense`.

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

Returns: This instance

Note:

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

opentrons

The model string for the pipette (e.g. p300_single_v1.5)

New in version 2.0.

property **mount**

Return the name of the mount this pipette is attached to

New in version 2.0.

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

Move the instrument.

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

New in version 2.0.

property **name**

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

New in version 2.0.

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

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

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

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

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

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

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

```
from opentrons import protocol_api

# metadata
metadata = {
    'protocolName': 'My Protocol',
    'author': 'Name <email@address.com>',
    'description': 'Simple paired pipette protocol,
    'apiLevel': '2.10'
```

opentrons

```
left_pipette = ctx.load_instrument('p300_single_gen2', 'left')

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

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

Note:

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

New in version 2.7.

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

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

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

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

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

Parameters:
- **location** (**types.Location** or **Well** to pick up a tip from.) – The location from which to pick up a tip.
- **presses** (*int*) – The number of times to lower and then raise the pipette when picking up a tip, to ensure a good seal (0 [zero] will result in the pipette hovering over the tip but not picking it up–generally not desireable, but could be used for dry-run).
- **increment** (*float*) – The additional distance to travel on each successive press (e.g.: if *presses=3* and *increment=1.0*, then the first press will travel down into the tip by 3.5mm, the second by 4.5mm, and the third by 5.5mm).

Returns: This instance

New in version 2.0.

reset_tipracks(*self*)

Reload all tips in each tip rack and reset starting tip

opentrons

The height to return a tip to its tiprack.

New in version 2.2.

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

If a tip is currently attached to the pipette, then it will return the tip to it's location in the tiprack.

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

Returns: This instance

Parameters: **home_after** – See the `home_after` parameter in **drop_tip**.

New in version 2.0.

property **speed**

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

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

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

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

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

New in version 2.0.

property **starting_tip**

The starting tip from which the pipette pick up

New in version 2.0.

property **tip_racks**

The tip racks that have been linked to this pipette.

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

New in version 2.0.

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

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

Parameters:

- **location** (**Well** or None) – If no location is passed, pipette will touch tip at current well's edges
- **radius** (*float*) – Describes the proportion of the target well's radius. When *radius=1.0*, the pipette tip will move to the edge of the target well; when *radius=0.5*, it will move to 50% of the

mm

- **speed** (*float*) – The speed for touch tip motion, in mm/s. Default: 60.0 mm/s, Max: 80.0 mm/s, Min: 20.0 mm/s

Raises:

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

Returns: This instance

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

New in version 2.0.

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

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

Parameters:

- **volume** – The amount of volume to aspirate from each source and dispense to each destination. If volume is a list, each volume will be used for the sources/targets at the matching index. If volumes is a tuple with two elements, like *(20, 100)*, then a list of volumes will be generated with a linear gradient between the two volumes in the tuple.
- **source** – A single well or a list of wells from where liquid will be aspirated.
- **dest** – A single well or a list of wells where liquid will be dispensed to.
- ****kwargs** – See below

Keyword Arguments:

- *new_tip* (`string`) –
  - ‘never’: no tips will be picked up or dropped during transfer
  - ‘once’: (default) a single tip will be used for all commands.
  - ‘always’: use a new tip for each transfer.
- *trash* (`boolean`) – If *True* (default behavior), tips will be dropped in the trash container attached this *Pipette*. If *False* tips will be returned to tiprack.
- *touch_tip* (`boolean`) – If *True*, a **touch_tip()** will occur following each **aspirate()** and **dispense()**. If set to *False* (default behavior), no **touch_tip()** will occur.
- *blow_out* (`boolean`) – If *True*, a **blow_out()** will occur following each **dispense()**, but only if the pipette has no liquid left in it. If set to *False* (default), no **blow_out()** will occur.
- *blowout_location* (`string`) –
  - ‘source well’: blowout excess liquid into source well
  - ‘destintation well’: blowout excess liquid into destination well
  - ‘trash’: blowout excess liquid into the trash

opentrons

pipette is empty, a **blow_out()** will occur into the trash
If *blow_out* is set to *False*, this parameter will be ignored

- *mix_before* (`tuple`) – The tuple, if specified, gives the amount of volume to **mix()** preceding each **aspirate()** during the transfer. The tuple is interpreted as (repetitions, volume).
- *mix_after* (`tuple`) – The tuple, if specified, gives the amount of volume to **mix()** after each **dispense()** during the transfer. The tuple is interpreted as (repetitions, volume).
- *disposal_volume* (`float`) – (**distribute()** only) Volume of liquid to be disposed off after distributing. When dispensing multiple times from the same tip, it is recommended to aspirate an extra amount of liquid to be disposed off after distributing.
- *carryover* (`boolean`) – If *True* (default), any *volume* that exceeds the maximum volume of this Pipette will be split into multiple smaller volumes.
- *gradient* (`lambda`) – Function for calculating the curve used for gradient volumes. When *volume* is a tuple of length 2, its values are used to create a list of gradient volumes. The default curve for this gradient is linear (lambda x: x), however a method can be passed with the *gradient* keyword argument to create a custom curve.

Returns: This instance

New in version 2.0.

property **trash_container**

The trash container associated with this pipette.

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

New in version 2.0.

property **type**

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

New in version 2.0.

property **well_bottom_clearance**

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

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

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

To change, set the corresponding attribute. For instance,

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

New in version 2.0.

opentrons

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

class opentrons.protocol_api.labware.**Labware**(*implementation: opentrons.protocols.context.labware.AbstractLabware*, *api_level: Optional[opentrons.protocols.api_support.types.APIVersion] = None*)

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

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

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

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

property **api_version**

New in version 2.0.

property **calibrated_offset**

New in version 2.0.

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

Accessor function used to navigate through a labware by column.

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

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

Returns: A list of column lists

New in version 2.0.

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

New in version 2.0.

columns_by_name(*self*) → Dict[str, List[opentrons.protocol_api.labware.Well]]

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

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

opentrons

property **highest_z**

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

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

New in version 2.0.

property **is_tiprack**

New in version 2.0.

property **load_name**

The API load name of the labware definition

New in version 2.0.

property **magdeck_engage_height**

New in version 2.0.

property **name**

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

New in version 2.0.

next_tip(*self*, *num_tips: int = 1*, *starting_tip: Union[opentrons.protocol_api.labware.Well, NoneType] = None*) → Union[opentrons.protocol_api.labware.Well, NoneType]

Find the next valid well for pick-up.

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

Parameters:
- **num_tips** (*int*) – target number of sequential tips in the same column
- **starting_tip** (`Well`) – The `Well` from which to start search. for an available tip.

Returns: the `Well` meeting the target criteria, or None

property **parameters**

Internal properties of a labware including type and quirks

New in version 2.0.

property **parent**

The parent of this labware. Usually a slot name.

New in version 2.0.

previous_tip(*self*, *num_tips: int = 1*) → Union[opentrons.protocol_api.labware.Well, NoneType]

Find the best well to drop a tip in.

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

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

New in version 2.0.

reset(*self*)

Reset all tips in a tiprack

New in version 2.0.

return_tips(*self*, *start_well: opentrons.protocol_api.labware.Well*, *num_channels: int = 1*)

Re-adds tips to the tip tracker

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

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

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

rows(*self*, **args*) → List[List[opentrons.protocol_api.labware.Well]]

Accessor function used to navigate through a labware by row.

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

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

Returns: A list of row lists

New in version 2.0.

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

New in version 2.0.

rows_by_name(*self*) → Dict[str, List[opentrons.protocol_api.labware.Well]]

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

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

Returns: Dictionary of Well lists keyed by row name

New in version 2.0.

set_calibration(*self*, *delta: opentrons.types.Point*)

Called by save calibration in order to update the offset on the object.

property **tip_length**

opentrons

A string fully identifying the labware.

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

New in version 2.0.

use_tips(*self*, *start_well: opentrons.protocol_api.labware.Well*, *num_channels: int = 1*)

Removes tips from the tip tracker.

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

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

well(*self*, *idx*) → opentrons.protocol_api.labware.Well

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

New in version 2.0.

wells(*self*, **args*) → List[opentrons.protocol_api.labware.Well]

Accessor function used to generate a list of wells in top -> down, left -> right order. This is representative of moving down *rows* and across *columns* (e.g. 'A1', 'B1', 'C1'...'A2', 'B2', 'C2')

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

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

Returns: Ordered list of all wells in a labware

New in version 2.0.

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

New in version 2.0.

wells_by_name(*self*) → Dict[str, opentrons.protocol_api.labware.Well]

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

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

Returns: Dictionary of well objects keyed by well name

New in version 2.0.

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

opentrons

The Well class represents a single well in a `Labware`

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

property **api_version**

New in version 2.0.

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

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

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

New in version 2.0.

center(*self*) → opentrons.types.Location

Returns: a Point corresponding to the absolute position of the center of the well relative to the deck (with the front-left corner of slot 1 as (0,0,0))

New in version 2.0.

property **depth**

The depth of a well in a labware.

New in version 2.9.

property **diameter**

New in version 2.0.

from_center_cartesian(*self*, *x: float*, *y: float*, *z: float*) → opentrons.types.Point

Specifies an arbitrary point in deck coordinates based on percentages of the radius in each axis. For example, to specify the back-right corner of a well at 1/4 of the well depth from the bottom, the call would be *from_center_cartesian(1, 1, -0.5)*.

No checks are performed to ensure that the resulting position will be inside of the well.

Parameters:
- **x** – a float in the range [-1.0, 1.0] for a percentage of half of the radius/length in the X axis
- **y** – a float in the range [-1.0, 1.0] for a percentage of half of the radius/width in the Y axis
- **z** – a float in the range [-1.0, 1.0] for a percentage of half of the height above/below the center

Returns: a Point representing the specified location in absolute deck

coordinates

New in version 2.8.

property **has_tip**

New in version 2.0.

property **length**

The length of a well, if the labware has square wells.

New in version 2.9.

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

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

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

New in version 2.0.

property **well_name**

New in version 2.7.

property **width**

The width of a well, if the labware has square wells.

New in version 2.9.

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

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

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

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

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

opentrons.protocol_api.labware.**load**(*load_name: str, parent: opentrons.types.Location, label: Union[str, NoneType] = None, namespace: Union[str, NoneType] = None, version: int = 1, bundled_defs: Union[Dict[str, ForwardRef('LabwareDefinition')], NoneType] = None, extra_defs: Union[Dict[str, ForwardRef('LabwareDefinition')], NoneType] = None, api_level: Union[opentrons.protocols.api_support.types.APIVersion, NoneType] = None*) → opentrons.protocol_api.labware.Labware

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

Parameters:
- **load_name** – A string to use for looking up a labware definition previously saved to disc. The definition file must have been saved in a known location
- **parent** – A `Location` representing the location where the front and left most point of the outside of labware is (often the front-left corner of a slot on the deck).
- **label** (*str*) – An optional label that will override the labware's display name from its definition
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.

opentrons

passed, these definitions will also be searched.

- **api_level** (*APIVersion*) – the API version to set for the loaded labware instance. The `Labware` will conform to this level. If not specified, defaults to `MAX_SUPPORTED_VERSION`.

`opentrons.protocol_api.labware.`**`load_from_definition`**(*definition: 'LabwareDefinition'*, *parent: opentrons.types.Location*, *label: Union[str, NoneType] = None*, *api_level: Union[opentrons.protocols.api_support.types.APIVersion, NoneType] = None*) → opentrons.protocol_api.labware.Labware

Return a labware object constructed from a provided labware definition dict

Parameters:

- **definition** – A dict representing all required data for a labware, including metadata such as the display name of the labware, a definition of the order to iterate over wells, the shape of wells (shape, physical dimensions, etc), and so on. The correct shape of this definition is governed by the "labware-designer" project in the Opentrons/opentrons repo.
- **parent** – A `Location` representing the location where the front and left most point of the outside of labware is (often the front-left corner of a slot on the deck).
- **label** (*str*) – An optional label that will override the labware's display name from its definition
- **api_level** (*APIVersion*) – the API version to set for the loaded labware instance. The `Labware` will conform to this level. If not specified, defaults to `MAX_SUPPORTED_VERSION`.

`opentrons.protocol_api.labware.`**`save_definition`**(*labware_def: 'LabwareDefinition'*, *force: bool = False*, *location: Union[pathlib.Path, NoneType] = None*) → None

Save a labware definition

Parameters:

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

`opentrons.protocol_api.labware.`**`select_tiprack_from_list_paired_pipettes`**(*tip_racks: List[opentrons.protocol_api.labware.Labware], p_channels: int, s_channels: int, starting_point: Union[opentrons.protocol_api.labware.Well, NoneType] = None*) → Tuple[opentrons.protocol_api.labware.Labware, opentrons.protocol_api.labware.Well]

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

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

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

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

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

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

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

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

Raises:

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

Returns:

opentrons

Modules

class opentrons.protocol_api.contexts.**TemperatureModuleContext**(*ctx: ProtocolContext*, *hw_module: modules.tempdeck.TempDeck*, *geometry: ModuleGeometry*, *at_version: APIVersion*, *loop: asyncio.AbstractEventLoop*)

An object representing a connected Temperature Module.

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

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

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

New in version 2.0.

property **api_version**

New in version 2.0.

await_temperature(*self*, *celsius: 'float'*)

Wait until module reaches temperature, in C.

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

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

New in version 2.3.

deactivate(*self*)

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

New in version 2.0.

property **geometry**

The object representing the module as an item on the deck

Returns: ModuleGeometry

New in version 2.0.

property **labware**

The labware (if any) present on this module.

New in version 2.0.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'int' = 1*) → 'Labware'

Specify the presence of a piece of labware on the module.

Parameters:

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

'opentrons' then 'custom_beta'

- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.

Returns: The initialized and loaded labware object.

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

New in version 2.0.

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

New in version 2.1.

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

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

Parameters:

- **definition** – The labware definition.
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.

Returns: The initialized and loaded labware object.

New in version 2.0.

load_labware_object(*self*, *labware: 'Labware'*) → 'Labware'

Specify the presence of a piece of labware on the module.

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

Returns: The properly-linked labware object

New in version 2.0.

set_temperature(*self*, *celsius: 'float'*)

Set the target temperature, in C.

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

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

New in version 2.0.

property **status**

The status of the module.

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

New in version 2.3.

property **target**

Current target temperature in C

New in version 2.0.

property **temperature**

class `opentrons.protocol_api.contexts.`**`MagneticModuleContext`**(*ctx: ProtocolContext*, *hw_module: modules.magdeck.MagDeck*, *geometry: ModuleGeometry*, *at_version: APIVersion*, *loop: asyncio.AbstractEventLoop*)

An object representing a connected Temperature Module.

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

New in version 2.0.

property **`api_version`**

New in version 2.0.

`calibrate`(*self*)

Calibrate the Magnetic Module.

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

New in version 2.0.

`disengage`(*self*)

Lower the magnets back into the Magnetic Module.

New in version 2.0.

`engage`(*self*, *height: 'Optional[float]' = None*, *offset: 'Optional[float]' = None*, *height_from_base: 'Optional[float]' = None*)

Raise the Magnetic Module's magnets.

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

- If neither `height`, `height_from_base` nor `offset` is specified, the magnets will raise to a reasonable default height based on the specified labware.
- The recommended way to adjust the height of the magnets is to specify `height_from_base`, which should be a distance in mm relative to the base of the labware that is on the magnetic module
- If `height` is specified, it should be a distance in mm from the home position of the magnets.
- If `offset` is specified, it should be an offset in mm from the default position. A positive number moves the magnets higher and a negative number moves the magnets lower.

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

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

New in version 2.1: The *height_from_base* parameter.

New in version 2.0.

property **`geometry`**

The object representing the module as an item on the deck

opentrons

property **labware**

The labware (if any) present on this module.

New in version 2.0.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'int' = 1*) → 'Labware'

Specify the presence of a piece of labware on the module.

Parameters:
- **name** – The name of the labware object.
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.

Returns: The initialized and loaded labware object.

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

New in version 2.0.

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

New in version 2.1.

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

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

Parameters:
- **definition** – The labware definition.
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.

Returns: The initialized and loaded labware object.

New in version 2.0.

load_labware_object(*self*, *labware: 'Labware'*) → 'Labware'

Load labware onto a Magnetic Module, checking if it is compatible

New in version 2.0.

property **status**

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

New in version 2.0.

class opentrons.protocol_api.contexts.**ThermocyclerContext**(*ctx: ProtocolContext*, *hw_module: modules.thermocycler.Thermocycler*, *geometry: ThermocyclerGeometry*, *at_version: APIVersion*, *loop: asyncio.AbstractEventLoop*)

An object representing a connected Temperature Module.

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

New in version 2.0.

opentrons

property **block_target_temperature**

Target temperature in degrees C

New in version 2.0.

property **block_temperature**

Current temperature in degrees C

New in version 2.0.

property **block_temperature_status**

New in version 2.0.

close_lid(*self*)

Closes the lid

New in version 2.0.

deactivate(*self*)

Turn off the well block temperature controller, and heated lid

New in version 2.0.

deactivate_block(*self*)

Turn off the well block temperature controller

New in version 2.0.

deactivate_lid(*self*)

Turn off the heated lid

New in version 2.0.

execute_profile(*self*, *steps: 'List[modules.ThermocyclerStep]'*, *repetitions: 'int'*, *block_max_volume: 'Optional[float]' = None*)

Execute a Thermocycler Profile defined as a cycle of **steps** to repeat for a given number of **repetitions**

Parameters:
- **steps** – List of unique steps that make up a single cycle. Each list item should be a dictionary that maps to the parameters of the **set_block_temperature()** method with keys 'temperature', 'hold_time_seconds', and 'hold_time_minutes'.
- **repetitions** – The number of times to repeat the cycled steps.
- **block_max_volume** – The maximum volume of any individual well of the loaded labware. If not supplied, the thermocycler will default to 25µL/well.

New in version 2.0.

property **geometry**

The object representing the module as an item on the deck

Returns: ModuleGeometry

New in version 2.0.

property **labware**

property **lid_position**

Lid open/close status string

New in version 2.0.

property **lid_target_temperature**

Target temperature in degrees C

New in version 2.0.

property **lid_temperature**

Current temperature in degrees C

New in version 2.0.

property **lid_temperature_status**

New in version 2.0.

load_labware(*self*, *name: 'str'*, *label: 'Optional[str]' = None*, *namespace: 'Optional[str]' = None*, *version: 'int' = 1*) → 'Labware'

Specify the presence of a piece of labware on the module.

Parameters:
- **name** – The name of the labware object.
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.
- **namespace** (*str*) – The namespace the labware definition belongs to. If unspecified, will search 'opentrons' then 'custom_beta'
- **version** (*int*) – The version of the labware definition. If unspecified, will use version 1.

Returns: The initialized and loaded labware object.

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

New in version 2.0.

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

New in version 2.1.

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

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

Parameters:
- **definition** – The labware definition.
- **label** (*str*) – An optional special name to give the labware. If specified, this is the name the labware will appear as in the run log and the calibration view in the Opentrons app.

Returns: The initialized and loaded labware object.

New in version 2.0.

load_labware_object(*self*, *labware: 'Labware'*) → 'Labware'

Specify the presence of a piece of labware on the module.

Parameters: **labware** – The labware object. This object should be already initialized and its parent should be

New in version 2.0.

open_lid(*self*)

Opens the lid

New in version 2.0.

set_block_temperature(*self*, *temperature: 'float'*, *hold_time_seconds: 'Optional[float]' = None*, *hold_time_minutes: 'Optional[float]' = None*, *ramp_rate: 'Optional[float]' = None*, *block_max_volume: 'Optional[float]' = None*)

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

Valid operational range yet to be determined.

Parameters:
- **temperature** – The target temperature, in °C.
- **hold_time_minutes** – The number of minutes to hold, after reaching `temperature`, before proceeding to the next command.
- **hold_time_seconds** – The number of seconds to hold, after reaching `temperature`, before proceeding to the next command. If `hold_time_minutes` and `hold_time_seconds` are not specified, the Thermocycler will proceed to the next command after `temperature` is reached.
- **ramp_rate** – The target rate of temperature change, in °C/sec. If `ramp_rate` is not specified, it will default to the maximum ramp rate as defined in the device configuration.
- **block_max_volume** – The maximum volume of any individual well of the loaded labware. If not supplied, the thermocycler will default to 25µL/well.

New in version 2.0.

set_lid_temperature(*self*, *temperature: 'float'*)

Set the target temperature for the heated lid, in °C.

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

New in version 2.0.

Useful Types and Definitions

class `opentrons.types.`**DeckSlotName**

Deck slot identifiers.

class `opentrons.types.`**Location**(*point: opentrons.types.Point, labware: Union[Labware, Well, str, ModuleGeometry, opentrons.protocols.api_support.labware_like.LabwareLike, None]*)

A location to target as a motion.

The location contains a **`Point`** (in Deck Coordinates) and possibly an associated **`Labware`** or **`Well`** instance.

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

Warning:

opentrons

specify the correct labware for the **point** attribute.

Warning:

The == operation compares both the position and associated labware. If you only need to compare locations, compare the **point** of each item.

move(*self*, *point: 'Point'*) → "'Location'"

Alter the point stored in the location while preserving the labware.

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

```
>>> loc = Location(Point(1, 1, 1), None)
>>> new_loc = loc.move(Point(1, 1, 1))
>>> assert loc_2.point == Point(2, 2, 2)  # True
>>> assert loc.point == Point(1, 1, 1)  # True
```

class opentrons.types.**Mount**

An enumeration.

class opentrons.types.**MountType**

An enumeration.

exception opentrons.types.**PipetteNotAttachedError**

An error raised if a pipette is accessed that is not attached

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

property **x**

Alias for field number 0

property **y**

Alias for field number 1

property **z**

Alias for field number 2

class opentrons.types.**TransferTipPolicy**

An enumeration.

Executing and Simulating Protocols

opentrons.execute: functions and entrypoint for running protocols

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

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

bound up in other internal server infrastructure) sources.

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

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

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

Parameters:

- **protocol_file** (*file-like*) – The protocol file to execute
- **protocol_name** (*str*) – The name of the protocol file. This is required internally, but it may not be a thing we can get from the protocol_file argument.
- **propagate_logs** (*bool*) – Whether this function should allow logs from the Opentrons stack to propagate up to the root handler. This can be useful if you're integrating this function in a larger application, but most logs that occur during protocol simulation are best associated with the actions in the protocol that cause them. Default: `False`
- **log_level** (*'debug', 'info', 'warning', or 'error'*) – The level of logs to emit on the command line.. Default: 'warning'
- **emit_runlog** – A callback for printing the runlog. If specified, this will be called whenever a command adds an entry to the runlog, which can be used for display and progress estimation. If specified, the callback should take a single argument (the name doesn't matter) which will be a dictionary (see below). Default: `None`
- **custom_labware_paths** – A list of directories to search for custom labware, or None. Ignored if the apiv2 feature flag is not set. Loads valid labware from these paths and makes them available to the protocol context.
- **custom_data_paths** – A list of directories or files to load custom data files from. Ignored if the apiv2 feature flag if not set. Entries may be either files or directories. Specified files and the non-recursive contents of specified directories are presented by the protocol context in **`ProtocolContext.bundled_data`**.

The format of the runlog entries is as follows:

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

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

Get the argument parser for this module

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

Parameters: **parser** – A parser to add arguments to.

Returns The parser with arguments added.

opentrons

ForwardRef('LabwareDefinition')]] = None) → opentrons.protocol_api.protocol_context.ProtocolContext

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

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

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

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

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

Parameters:

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

Returns opentrons.protocol_api.ProtocolContext: The protocol context.

`opentrons.execute.`**main**() → int

Handler for command line invocation to run a protocol.

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

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

opentrons.simulate: functions and entrypoints for simulating protocols

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

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

emit(*self*, *record*)

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

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

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

level to scrape, and the opentrons broker object to subscribe to.

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

property **commands**

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

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

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

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

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

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

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

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

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

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

Get the argument parser for this module

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

Parameters:	**parser** – A parser to add arguments to. If not specified, one will be created.
Returns argparse.ArgumentParser:	The parser with arguments added.

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

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

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

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

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

Parameters:

- **version** – The API version to use. This must be lower than **opentrons.protocol_api.MAX_SUPPORTED_VERSION**. It may be specified either as a string (`'2.0'`) or as a **protocols.types.APIVersion** (`APIVersion(2, 0)`).

preparation for a beta feature and is best not used.

- **bundled_data** – If specified, a mapping from filenames to contents for data to be available in the protocol from **`ProtocolContext.bundled_data`**.
- **extra_labware** – If specified, a mapping from labware names to labware definitions for labware to consider in the protocol in addition to those stored on the robot. If this is an empty dict, and this function is called on a robot, it will look in the 'labware' subdirectory of the Jupyter data directory for custom labware.
- **hardware_simulator** – If specified, a hardware simulator instance.

Returns The protocol context.

opentrons.protocol_api.ProtocolContext:

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

Run the simulation

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

Simulate the protocol itself.

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

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

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

The return value is the run log, a list of dicts that represent the commands executed by the robot; and either the contents of the protocol that would be required to bundle, or `None`.

Each dict element in the run log has the following keys:

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

Parameters:

- **protocol_file** (*file-like*) – The protocol file to simulate.
- **file_name** (*str*) – The name of the file
- **custom_labware_paths** – A list of directories to search for custom labware, or None. Ignored if the apiv2 feature flag is not set. Loads valid labware from these paths and makes them available to the protocol context.
- **custom_data_paths** – A list of directories or files to load custom data files from. Ignored if the apiv2 feature flag if not set. Entries may be either files or directories. Specified files and the non-recursive

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

Returns: A tuple of a run log for user output, and possibly the required data to write to a bundle to bundle this protocol. The bundle is only emitted if bundling is allowed (see **`allow_bundling()`**) and this is an unbundled Protocol API v2 python protocol. In other cases it is None.

Deck Coordinates

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

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

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