A beginner's approach to OT-2 APIv2

June 10, 2020

Table of Contents

1 Introduction

This workshop aims at making the OT-2 APIv2 as approachable as possible for people like me who have limited experience in programming. The workshop material below is in someway just a skimmed walkthrough of the Opentrons OT-2 APIv2 docs. However, this is not meant to be a OT-2 programming 101 course and so it does not follow the official order nor their recommendations. OT-2 users who wish to make the most out of the robot are encouraged to peruse the docs.

In the rest of the document, API refers to APIv2.

To summarize, our approach to the API is:

First you make it work, then you make it work better

Part 1: The basics, for start to finish

1. Build a virtual environment (a sandbox) to play around with features of the API
2. Unit test the labware, wells, pipettes and commands and get instant readouts
3. Generate a human-trackable unit action, i.e. one single liquid transfer
4. Loop that unit action to scale up the experiment
5. Simulate the draft in the virtual environment, get the simulated output and check whether it is correct
6. Remove the virtual environment and convert the draft into a proper script
7. Do a final confirmation by simulating the script and exporting its formal runlog (**important for experiment documentation**)

Part 2: Customizations, how to make our experiments really work, and how to make our lives easier

1. Customize the transfer process
2. Use building block commands to achieve control precision
3. Write reusable scripts to avoid creating new scripts from scratch every time
4. Semi-integrate with Microsoft Excel so outputs from other instruments could be used to direct transfer processes

2 Disclaimer

1. I have limited experience in programming and received little formal training. Coding practices may be unorthodox and even heretic in the eyes of properly trained programmers.
2. Methods of execution are in most cases suboptimal in performance. (see Additional Remarks #1)
3. Please forgive my ocassional loss of professionalism in language.

3 Intended Outcomes

By the end of the workshop, we hope participants would be able to:

1. Read the documentation on APIv2, and know where to look for more information, and where to seek help if needed
2. Install the Opentrons module
3. Import the opentrons library to initialize a protocol object, create labware objects and pipettes within it and retrieve their information
4. Debug the code using the variable explorer and the console
5. Access wells of plates / racks
6. Instruct the pipettes to carry out liquid movement using pipette.transfer() and configure it with additional parameters
7. Create a custom pipetting step using building block commands
8. Use loops, lists and dictionaries together to scale up pipetting steps
9. Simulate the protocol in a PC and then converting it to a robot-executable one
10. Export simulate results to a readable-file.
11. Write the framework of practically useful protocols for automatable experiments
12. Use Excel to create list of parsable commands to be copied and pasted into a protocol

4 (Part I) Preparation: System Setup

4.1 Install the IDE

It is best to make use of an Integrated Development Environment (IDE) when writing codes. My personal experience suggests that Spyder is most friendly to Python beginners and its Variable Explorer is tremendously helpful, both in debugging and in understanding how the API works. Some other workshops recommend Jupyter or PyCharm. Participants are welcomed to stick with the IDEs they are most comfortable with.

The relatively fool-proof way to get Spyder is to download and install Anaconda. For advanced users, the accompanying site packages like pandas and NumPy comes in handy for interfacing experiment results and experiment automation. (This will not be covered here)

4.2 Install the opentrons module

Once Anaconda is installed, participants should open Spyder, and install the opentrons module / library, by typing in the IPython console:

```
pip install opentrons
```

To check if the opentrons is correctly installed, participants should try and see if they could get the module version from the console.

[1]:
```
import opentrons
print(opentrons.__version__)
```

```
3.18.0
```

Failure to do so should give an error message of `ModuleNotFoundError: No module named 'opentrons'`

4.3 Keeping the opentrons module up-to-date

Opentrons update their API and library from time to time and it should happen together with OT-2 App and robot firmware upgrade. Always check if the opentrons module is the latest, and if needed, upgrade the opentrons module on the computer by typing in the terminal

```
pip install opentrons --upgrade
```

5 Setting up a virtual environment for simulating a robot

The following section corresponds to "Using Python For Protocols: Simulating your scripts" from the API docs.

Let's start by typing the following into the Spyder Editor.

[2]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults

Loading json containers…
Json container file load complete, listing database
Found 0 containers to add. Starting migration…
Database migration complete!
```

The `protocol` object is the starting point of all robot related actions, like choosing the pipette and putting labware on deck. It also stores the simulated output of the robot. To inspect its functions and attributes, run `dir(protocol)` in the IPython console.

6 Labware objects

The following section corresponds to "Labware" from the Opentrons docs.

6.1 Putting labware objects on the deck

Opentrons' definition of labware includes both the mult-well plates/racks/reserviors and the tip racks. They are loaded onto the robot via the function:
`labware_name = protocol.load_labware(load_name, location)`
- load_name: what is the labware, **as called by Opentrons**?
To look up this information, go to the Opentrons official labware page. In the following example we will use a 200 µL 96-well flat bottom plate.
The `load_name` of this labware is `corning_96_wellplate_360ul_flat`.
See Additional Remarks 3 for using a generic versus specific `load_name`.

- slot: which of the slot (1-11) are we placing it?
  This argument can be input as an integer or a string.

[3]:
```
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
```

Now if we inspect the variable explorer we should see the an object of `plate` being created. Any labware objects that we load by this command exists as a virtual object in the simulated environment.

[4]:
```
plate
```

[4]:
```
Corning 96 Well Plate 360 µL Flat on 1
```

Again, we can double click on the object in the variable explorer or use `dir(plate)` to look at the attributes of this plate.

We also need to put tips and tip racks onto the deck (say slot 2). So we will add and run the following lines.

[5]:
```
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
```

It should also appear in the variable explorer, can be checked just by typing the variable in the console.

[6]:
```
tip_rack
```

[6]:
```
Opentrons 96 Tip Rack 300 µL on 2
```

6.2 Getting wells within a labware

Before adding the pipettes let's briefy talk about how to get the wells in a labware object. This applies to whatever labware we are using, whether it's a multi-well plate or a tube rack or even a tip rack.

The "Labware: Accessor Methods" section on the Opentrons docs provide multiple ways to get the wells on labware, but we will only introduce two methods, which are usually sufficient to deal with most situations.

The conventional or user-firendly way: `labware.wells_by_name()[well_name]`

[7]:
```
# get a single well
plate.wells_by_name()['A6']
```

[7]:
```
A6 of Corning 96 Well Plate 360 µL Flat on 1
```

[8]:
```
# get a single well and assign it to a variable
sample_well = plate.wells_by_name()['A6']
sample_well
```

[8]:
```
A6 of Corning 96 Well Plate 360 µL Flat on 1
```

The other way is that get the wells by their **well indices**. This can come in handy in some other situations.

Method: `labware.wells()[well_index]`

Well index for a 96-well format:

	1	2	3	4	5	6	7	8	9	10	11	12
A	0	8	16	24	32	40	48	56	64	72	80	88
B	1	9	17	25	33	41	49	57	65	73	81	89
C	2	10	18	26	34	42	50	58	66	74	82	90
D	3	11	19	27	35	43	51	59	67	75	83	91
E	4	12	20	28	36	44	52	60	68	76	84	92
F	5	13	21	29	37	45	53	61	69	77	85	93
G	6	14	22	30	38	46	54	62	70	78	86	94
H	7	15	23	31	39	47	55	63	71	79	87	95

Well index for a 24-well format (whether tube rack or 24-well plate):

	1	2	3	4	5	6
A	0	4	8	12	16	20
B	1	5	9	13	17	21
C	2	6	10	14	18	22
D	3	7	11	15	19	23

[9]:
```
# Get well A6 in 96-well plate
plate.wells()[40]
```

[9]:
```
A6 of Corning 96 Well Plate 360 µL Flat on 1
```

[10]:
```
# Get well A6 in 24-well plate
plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 3)
plate2.wells()[20]
```

[10]:
```
A6 of Corning 24 Well Plate 3.4 mL Flat on 3
```

One advantage of using well index instead of well name, is that we can easily get a **continuous** list of wells by inputing a "slice" of the list.

[11]:
```
plate.wells()[0:12]
```

[11]:
```
[A1 of Corning 96 Well Plate 360 µL Flat on 1,
 B1 of Corning 96 Well Plate 360 µL Flat on 1,
 C1 of Corning 96 Well Plate 360 µL Flat on 1,
 D1 of Corning 96 Well Plate 360 µL Flat on 1,
 E1 of Corning 96 Well Plate 360 µL Flat on 1,
 F1 of Corning 96 Well Plate 360 µL Flat on 1,
 G1 of Corning 96 Well Plate 360 µL Flat on 1,
 H1 of Corning 96 Well Plate 360 µL Flat on 1,
 A2 of Corning 96 Well Plate 360 µL Flat on 1,
 B2 of Corning 96 Well Plate 360 µL Flat on 1,
 C2 of Corning 96 Well Plate 360 µL Flat on 1,
 D2 of Corning 96 Well Plate 360 µL Flat on 1]
```

Note that we **cannot** use these methods to get mulitple specific wells

[12]:
```
# Wrong way to get multiple wells by the .wells_by_name()
plate.wells_by_name()['A1', 'B2', 'C3']
```

```
---------------------------------------------------

KeyError         Traceback (most recent call last)

<ipython-input-12-ca2a01c16800> in <module>
  1 # Wrong way to get multiple wells by the .wells_by_name()
----> 2 plate.wells_by_name()['A1', 'B2', 'C3']


KeyError: ('A1', 'B2', 'C3')
```

[13]:
```
# Wrong way to get multiple wells by the .wells()
plate.wells()[1, 2, 3]
```

```
---------------------------------------------------

TypeError        Traceback (most recent call last)

<ipython-input-13-b3439485b881> in <module>
  1 # Wrong way to get multiple wells by the .wells()
----> 2 plate.wells()[1, 2, 3]
```

```
TypeError: list indices must be integers or slices, not tuple
```

7 Setting up pipettes

The following section corresponds to "Pipettes: Loading A Pipette" from the API docs.

The way to set up a pipette is to call the function: `pipette_name = protocol.load_instrument(instrumnt_name, mount, tip_racks)`

Refer to the "Pipettes: Pipette Models" for what instrument_name corresponds to what pipette.

Note 1: The argument `tip_racks` must be a **list**, this is a mistake I often make.

Note 2 : **Always** load the tip racks onto the virtual deck beforehand and **associate then those tip racks with the pipette** when we set up the pipette.

[14]:
```
# load a P300S on the right mount

r_pipette = protocol.load_instrument(
    'p300_single', # instrument_name
    'right', # mount
    tip_racks=[tip_rack] # tip_racks as list, note that we are re-using the␣
 ↪tip_rack object loaded before
)

r_pipette
```

[14]:
```
<InstrumentContext: p300_single_v1 in RIGHT>
```

[15]:
```
# Similarily, we can load a P10M on the left mount
# Remember to load a compatible tip rack first.

tip_rack_10 = protocol.load_labware('opentrons_96_tiprack_10ul', 4)

l_pipette = protocol.load_instrument(
    'p10_multi',
    'left',
    tip_racks=[tip_rack_10]
)

l_pipette
```

[15]:
```
<InstrumentContext: p10_multi_v1 in LEFT>
```

For the following examples we will stick with the P300S on the right mount only.

8 Getting readouts from the console

Basically by this time we have configured the robot properly. We can then instruct the pipette to take actions.

Before we move on to introduce the pipette commands, we will first include the following lines at the end of the script:

[16]:
```
print(*protocol.commands(), sep = '\n')
```

This line prints all the commands from the robot and will help us debug what is going on inside the script.

By this time, our script should look like the following:

[17]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

#%% Robot set up

# we slightly rearranged the locations
plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 2)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

r_pipette = protocol.load_instrument(
    'p300_single', # instrument_name
    'right', # mount
    tip_racks=[tip_rack] # tip_racks as list, note that we are re-using the␣
 ↪tip_rack object loaded before
)

#%% Robot actions

print(*protocol.commands(), sep = '\n')
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults
```

There is nothing from the console yet because we haven't assigned any commands to the pipettes.

9 Executing a simple liquid transfer

The following section corresponds to "Complex commands: Transfer" from the API docs.

We will only introduce the `pipette.transfer()` command since it is most versatile. There is really no need to use `pipette.distribute()` or `pipette.consolidate()` since `pipette.transfer()`

covers those functions already, unless it's for script readability.

The function should be called as follow: `pipette.transfer(volume, source, dest)`

Note: for this notebook the `print(protocol.commands())` function and the `protocol.clear_commands()` functions are called in every cell. This is not necessary when debugging the script in Spyder

9.1 "one-to-one" transfer

[18]:
```
r_pipette.transfer(200, # volume in µL
                   plate2.wells_by_name()['A1'], # source well
                   plate1.wells_by_name()['A1'] # dest well
                  )

print(*protocol.commands(), sep = '\n')
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

The output lines above are the instructions that the robot will execute, written in a human readable format. This is the log of the simuation and we should use that to check whether we wrote the script correctly.

The first line "Transferring … Flat on 1" is actually a section header that encapsulates all the subcommands (or building block commands) that follow it. We will introduce building commands later.

We can run another transfer step and pipette another 200 µL, but this time to well A2 on the 96-well plate.

[19]:
```
# a second transfer step

r_pipette.transfer(200, # volume in µL
                   plate2.wells_by_name()['A1'], # source well
                   plate1.wells_by_name()['A2'] # dest well
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
```

```
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

Reading the output above we take note of two things:

1. Two output commands are displayed instead of one. This is because the robot keeps on remembering new instructions that we have given it. This is usually not a problem when debugging in Spyder but one should always watch out for unintended duplciation of pipetting steps while writing the script.

2. We notice that in the 1st step, the pipette picks the tip from position A1 of the rack and in the 2nd step, it does so from position B1. This is the perk of associating a tip rack to the pipette during the `load_instrument()` function, i.e., the robot automatically keep tracks of tip usage (column by column from A1 to A8, B1 to B8…)

9.2 Large volumes

If we need to transfer, say, 400 µL but we only have a P300S, the `pipette.transfer()` function automatically does the calculation and performs two transfer steps of 200 µL each.

[20]:
```
r_pipette.transfer(400, # volume in µL
                   plate2.wells_by_name()['A1'], # source well
                   plate1.wells_by_name()['A2'] # dest well
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 400.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

9.3 Scaling up transfer by the "one-to-many", "many-to-one", or "many-to-many" transfer modes

The `pipette.transfer()` function is extremely versatile. See the following examples.

[21]:
```
# one-to-many
r_pipette.transfer(200,
                   plate2.wells_by_name()['A1'],
                   [
                       plate1.wells_by_name()['A1'],
                       plate1.wells_by_name()['A2'],
                       plate1.wells_by_name()['A3'] # instead of a single well,␣
 ↪pass a list of destination wells
                   ] # notice where the open and close brackets are
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

[22]:
```
# many-to-one

r_pipette.transfer(200,
                   [
                       plate2.wells_by_name()['A1'],
                       plate2.wells_by_name()['A2'],
                       plate2.wells_by_name()['A3'] # instead of a single well,␣
 ↪pass a list of destination wells
                   ], # again, notice where the open and close brackets are
                  plate1.wells_by_name()['A1']
                  )
```

```
print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A2 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A3 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

[23]:
```
# many-to-many
# A1 -> B1, A2 -> B2, A3 -> B3,

r_pipette.transfer(200,
                   [
                       plate1.wells_by_name()['A1'],
                       plate1.wells_by_name()['A2'],
                       plate1.wells_by_name()['A3'] # a list of source wells
                   ],
                   [
                       plate1.wells_by_name()['B1'],
                       plate1.wells_by_name()['B2'],
                       plate1.wells_by_name()['B3'] # a list of destination wells
                   ]
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from F1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
```

```
Dispensing 200.0 uL into B2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from A3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

When using many-to-many transfer mode, make sure that we have **equal numbers of source and destination wells**.

[24]:
```
# many-to-many, failed example
# A1 -> B1, A2 -> B2, A3 -> ?,

r_pipette.transfer(200,
                   [
                       plate1.wells_by_name()['A1'],
                       plate1.wells_by_name()['A2'],
                       plate1.wells_by_name()['A3'] # 3 source wells
                   ],
                   [
                       plate1.wells_by_name()['B1'],
                       plate1.wells_by_name()['B2'] # but only 2 destination wells
                   ]
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()

# Look at error below:
```

```
        ---------------------------------------------------

        ValueError       Traceback (most recent call last)

        <ipython-input-24-e59363f8acf2> in <module>
         10                   [
         11                       plate1.wells_by_name()['B1'],
    ---> 12                       plate1.wells_by_name()['B2'] # but only 2␣
↪destination wells
         13                   ]
         14                  )


        D:\Anaconda\lib\site-packages\opentrons\protocol_api\util.py in␣
↪decorated(*args, **kwargs)
        608                 do_publish(
```

```
      609                     broker, command, f, 'before', None, meta, *args,␣
↪**kwargs)
   --> 610             res = f(*args, **kwargs)
      611             if after:
      612                 do_publish(


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\util.py in␣
↪_check_version_wrapper(*args, **kwargs)
      347                     f'must increase your API version to {added_in}␣
↪to '
      348                     'use this functionality.')
   --> 349             return decorated_obj(*args, **kwargs)
      350
      351         return _check_version_wrapper


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\instrument_context.
↪py in transfer(self, volume, source, dest, trash, **kwargs)
     1001                                         self.api_version,␣
↪kwargs['mode'],
     1002                                         transfer_options)
   -> 1003         self._execute_transfer(plan)
     1004         return self
     1005


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\instrument_context.
↪py in _execute_transfer(self, plan)
     1005
     1006     def _execute_transfer(self, plan: transfers.TransferPlan):
   -> 1007         for cmd in plan:
     1008             getattr(self, cmd['method'])(*cmd['args'],␣
↪**cmd['kwargs'])
     1009


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪__iter__(self)
      426         yield from {TransferMode.CONSOLIDATE: self._plan_consolidate,
      427                     TransferMode.DISTRIBUTE: self._plan_distribute,
   --> 428                     TransferMode.TRANSFER: self._plan_transfer}[self.
↪_mode]()
      429         if self._strategy.new_tip == types.TransferTipPolicy.ONCE:
      430             if self._strategy.drop_tip_strategy == DropTipStrategy.
↪RETURN:
```

```
       D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪_plan_transfer(self)
       466         # reform source target lists
       467         sources, dests = self._extend_source_target_lists(
   --> 468             self._sources, self._dests)
       469         plan_iter = self._expand_for_volume_constraints(
       470             self._volumes, zip(sources, dests),


       D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪_extend_source_target_lists(sources, targets)
       502             if len(sources) % len(targets) != 0:
       503                 raise ValueError(
   --> 504                     'Source and destination lists must be divisible')
       505             targets = [target for target in targets
       506                        for i in range(int(len(sources)/
↪len(targets)))]


       ValueError: Source and destination lists must be divisible
```

[25]:
```
# This cell resets the robot from the error, this is not required in Spyder

from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

#%% Robot set up

# we slightly rearranged the locations
plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 2)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

r_pipette = protocol.load_instrument(
    'p300_single', # instrument_name
    'right', # mount
    tip_racks=[tip_rack] # tip_racks as list
)
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults
```

If we arrange the wells such that they are continuous along columns, using well indices would come in really handy.

[26]:
```
# many-to-many, along a column
# A1 -> A2,
# B1 -> B2,
# C1 -> C2,
# D1 -> D2

r_pipette.transfer(200,
                   plate1.wells()[0:4],
                   plate1.wells()[8:12]
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from C1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into C2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from D1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into D2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

10 Scaling up transfers using a for-loop

We can now repeat a single one-to-one transfer step over and over again. This is the biggest advantage of programming through the API, compared to the Protocol Designer.

[27]:
```
protocol.clear_commands()

for i in range(3):
    r_pipette.transfer(200, # volume in µL
                       plate2.wells_by_name()['A1'], # source well
                       plate1.wells_by_name()['A2'] # dest well
                      )
```

```
print(*protocol.commands(), sep = '\n')
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can see that the identical action is repeated three times but every time a new tip is used.

Repeating the identical step over and over again isn't very useful, however. Very often we want to do a one-to-many or many-to-many transfer. We can modify the scripts above to do the following:

[28]:
```
# one-to-many, essentially a pipette.distribute() process
# A1 -> B1, B2, B3, B4

protocol.clear_commands()

dest_wells = ['B1', 'B2', 'B3', 'B4']

for dest_well in dest_wells:
    r_pipette.transfer(200,
                       plate1.wells_by_name()['A1'],
                       plate1.wells_by_name()[ dest_well ] # dynamic dest well␣
 ↪that changes on every loop
                      )

print(*protocol.commands(), sep = '\n')
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from F1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B3 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from G1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B4 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from H1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B4 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

[29]:
```python
# many-to-many
# A1 -> B1,
# A2 -> B2,
# A3 -> B3,
# A4 -> B4

protocol.clear_commands()

source_wells = ['A1', 'A2', 'A3', 'A4']
dest_wells = ['B1', 'B2', 'B3', 'B4']

for i in range(4):
    r_pipette.transfer(200,
```

```
                    plate1.wells_by_name()[ source_wells[i] ], # dynamic␣
↪source well that changes on every loop
                    plate1.wells_by_name()[ dest_wells[i] ] # dynamic dest␣
↪well that changes on every loop
                   )

print(*protocol.commands(), sep = '\n')
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A2 of Corning 96 Well Plate 360 µL Flat on 1 to B2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from B2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A3 of Corning 96 Well Plate 360 µL Flat on 1 to B3 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from C2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A4 of Corning 96 Well Plate 360 µL Flat on 1 to B4 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from D2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A4 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B4 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

10.1 Nested for-loops example

The above examples are equivalents of the `pipette.transfer()` function, but there is more that we can achieve with for-loops.

Consider the following example. We wish to pipette:

plate 1 well A1 to plate 1 well A1, plate 1 well A2 to plate 1 well A2, plate 1 well A3 to plate 1

well A3, plate 1 well A4 to plate 1 well A4,

and then repeat the same process but substituting plate 1 with plate 2.

[30]:
```
# many-to-many,
# for plate 1 and plate 2
# A1 -> B1,
# A2 -> B2,
# A3 -> B3,
# A4 -> B4

protocol.clear_commands()

source_wells = ['A1', 'A2', 'A3', 'A4']
dest_wells = ['B1', 'B2', 'B3', 'B4']

for plate in [plate1, plate2]:

    # the for-loop beneath will run twice
    for i in range(4):
        r_pipette.transfer(200,
                           plate.wells_by_name()[ source_wells[i] ],
                             # note that it is now a variable "plate" instead of␣
 ↪the  object "plate1"
                           plate.wells_by_name()[ dest_wells[i] ]
                          )

print(*protocol.commands(), sep = '\n')
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from E2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A2 of Corning 96 Well Plate 360 µL Flat on 1 to B2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from F2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A3 of Corning 96 Well Plate 360 µL Flat on 1 to B3 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from G2 of Opentrons 96 Tip Rack 300 µL on 3
```

```
Aspirating 200.0 uL from A3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B3 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A4 of Corning 96 Well Plate 360 µL Flat on 1 to B4 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from H2 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A4 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B4 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to B1 of
Corning 24 Well Plate 3.4 mL Flat on 2
Picking up tip from A3 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A2 of Corning 24 Well Plate 3.4 mL Flat on 2 to B2 of
Corning 24 Well Plate 3.4 mL Flat on 2
Picking up tip from B3 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A2 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into B2 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A3 of Corning 24 Well Plate 3.4 mL Flat on 2 to B3 of
Corning 24 Well Plate 3.4 mL Flat on 2
Picking up tip from C3 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A3 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into B3 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A4 of Corning 24 Well Plate 3.4 mL Flat on 2 to B4 of
Corning 24 Well Plate 3.4 mL Flat on 2
Picking up tip from D3 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A4 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dispensing 200.0 uL into B4 of Corning 24 Well Plate 3.4 mL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

11 Note on using 8-channel pipettes

8-channel pipettes deal with 8 wells at the same time so for a `pipette.transfer` we can pass in a list of 8 well objects as the argument for `location`. We can also just assume the 8-channel pipette as a single channel pipette and use the top channel as the pointer.

Example:

[31]:
```
# Alternative 1: specifying 8 wells.

from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 2)

r_pipette = protocol.load_instrument(
    'p300_multi', # we use a 8-channel pipette here
    'right',
    tip_racks=[tip_rack]
)
r_pipette.transfer(200,
                   plate1.wells()[0:8], # wells A1-H1, or we can use plate1.
 ↪columns_by_name()['1']
                   plate1.wells()[8:16] # wells A2-H2, or we can use plate1.
 ↪columns_by_name()['2']
                      )
print(*protocol.commands(), sep = '\n')
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults

Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 2
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

[32]:
```
# Alternative 2: specifying the top well only

from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
```

```
r_pipette = protocol.load_instrument(
    'p300_multi', # we use a 8-channel pipette here
    'right',
    tip_racks=[tip_rack]
)
r_pipette.transfer(200,
                   plate1.wells()[0], # well A1
                   plate1.wells()[8] # well A2
                  )
print(*protocol.commands(), sep = '\n')
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults

Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 2
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can see that the there are no differences between the two methods. Usually, it is best to go for "specifying a single well". It works more relaibly in case you need to use a 8-channel pipette with a 384-well plate.

However, if the for-loop is used, we **must not loop through the groups of wells inside the `pipette.transfer()` function**

[33]:
```
# Wrong example of for-loop together with 8-channel pipette.

from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 2)

r_pipette = protocol.load_instrument(
    'p300_multi', # we use a 8-channel pipette here
    'right',
    tip_racks=[tip_rack]
)

for i in [8, 9 ,10, 11, 12, 13, 14, 15, 16]:
    r_pipette.transfer(200,
                       plate1.wells()[0],
                       plate1.wells()[i] # the intention may be wells in column␣
 ↪'2', but this
```

```
                    )

print(*protocol.commands(), sep = '\n')
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults
```

```
        ---------------------------------------------

        RuntimeError     Traceback (most recent call last)

        <ipython-input-33-338aec60f129> in <module>
         16     r_pipette.transfer(200,
         17                        plate1.wells()[0],
    ---> 18                        plate1.wells()[i] # the intention may be␣
↪wells in column '2', but this
         19                       )
         20


        D:\Anaconda\lib\site-packages\opentrons\protocol_api\util.py in␣
↪decorated(*args, **kwargs)
        608                 do_publish(
        609                     broker, command, f, 'before', None, meta, *args,␣
↪**kwargs)
    --> 610             res = f(*args, **kwargs)
        611             if after:
        612                 do_publish(


        D:\Anaconda\lib\site-packages\opentrons\protocol_api\util.py in␣
↪_check_version_wrapper(*args, **kwargs)
        347                     f'must increase your API version to {added_in}␣
↪to '
        348                     'use this functionality.')
    --> 349             return decorated_obj(*args, **kwargs)
        350
        351         return _check_version_wrapper


        D:\Anaconda\lib\site-packages\opentrons\protocol_api\instrument_context.
↪py in transfer(self, volume, source, dest, trash, **kwargs)
       1000         plan = transfers.TransferPlan(volume, source, dest, self,␣
↪max_volume,
       1001                                       self.api_version,␣
↪kwargs['mode'],
```

```
   -> 1002                                          transfer_options)
      1003         self._execute_transfer(plan)
      1004         return self


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪__init__(self, volume, sources, dests, instr, max_volume, api_version, mode,␣
↪options)
       383         # list of Wells into a 1 dimensional list of Wells
       384         if self._instr.hw_pipette['channels'] > 1:
   --> 385             sources, dests = self._multichannel_transfer(sources,␣
↪dests)
       386         else:
       387             if isinstance(sources, List) and isinstance(sources[0],␣
↪List):


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪_multichannel_transfer(self, s, d)
       823             if self._is_valid_row(well):
       824                 new_dst.append(well)
   --> 825         self._check_valid_well_list(new_dst, 'target', d)
       826         return new_src, new_dst
       827


      D:\Anaconda\lib\site-packages\opentrons\protocol_api\transfers.py in␣
↪_check_valid_well_list(self, well_list, id, old_well_list)
       784         if self._api_version >= APIVersion(2, 2) and len(well_list)␣
↪< 1:
       785             raise RuntimeError(
   --> 786                 f"Invalid {id} for multichannel transfer:␣
↪{old_well_list}")
       787
       788     def _multichannel_transfer(self, s, d):


      RuntimeError: Invalid target for multichannel transfer: [B2 of Corning␣
↪96 Well Plate 360 µL Flat on 1]
```

12 Finalizng the script for formal simulation and run

It should be noted that we have so far simulated our pipettes, labware and commands in a virtual environment, but since the `protocol` object we created was virtual, it cannot be transferred to an actual robot for execution. We will have to make some modifications on our script to make it a proper script.

12.1 Modifying the script to make it executable by OT-2

Say we have a simulated script below:

[ ]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 2)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

r_pipette = protocol.load_instrument(
    'p300_single',
    'right',
    tip_racks=[tip_rack]
)

r_pipette.transfer(200,
                   plate2.wells_by_name()['A1'],
                   plate1.wells_by_name()['A1']
                  )

r_pipette.transfer(150,
                   plate2.wells_by_name()['A1'],
                   plate1.wells_by_name()['A2']
                  )

print(*protocol.commands(), sep = '\n')
```

If we look at an example script as shown on the Opentrons Docs, we will notce a few key differences.

[ ]:
```
from opentrons import protocol_api

# metadata
metadata = {
    'protocolName': 'My Protocol',
    'author': 'Name <email@address.com>',
    'description': 'Simple protocol to get started using OT2',
    'apiLevel': '2.2'
}

# protocol run function. the part after the colon lets your editor know
# where to look for autocomplete suggestions
def run(protocol: protocol_api.ProtocolContext):

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

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

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

1. There is a line of `from opentrons import protocol_api`, and there are no lines related to `opentrons.simulate`, or `print(protocol.commands())`.
2. There is a metadata dictionary This is mostly for Opentrons to collect your personal data. You could leave everything out but you **must at least keep the 'apiLevel'**
3. Everything related to the robot (labware, pipettes and commands) are wrapped under a function `run()`

So to make the script executable on a robot, we have to make the above changes.

Tip: instead of deleting lines, **comment them out** by highlighting the text and press `Ctrl` + `1` in Spyder.
Caution: wrapping codes with a function means that they have to be **indented**. Highlight the code and press the `Tab` key will do the job. To unindent, highlight the text and press `Shift` + `Tab`.

If we make the above changes, our script will now look like the following:

[ ]:
```
# from opentrons import simulate
# protocol = simulate.get_protocol_api(version = '2.4')

from opentrons import protocol_api

metadata = {
    'apiLevel': '2.4'
}

def run(protocol: protocol_api.ProtocolContext):

    plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 2)
    tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

    r_pipette = protocol.load_instrument(
        'p300_single',
        'right',
        tip_racks=[tip_rack]
    )
```

```
    r_pipette.transfer(200,
                plate2.wells_by_name()['A1'],
                plate1.wells_by_name()['A1']
               )

    r_pipette.transfer(150,
                plate2.wells_by_name()['A1'],
                plate1.wells_by_name()['A2']
               )

# print(*protocol.commands(), sep = '\n')
```

In this format, our script is now ready to be executed on a robot.

But hold on, we should keep a simulated log for the experiment. If the protocol went awry, we can inspect a simulated log file to figure out whether we made mistakes in the code. However, after making these changes we cannot directly run the script in Spyder and get readouts, since the script is not designed to.

12.2 Protocol simulation - as instructed by Opentrons

The following section corresponds to "Using Python For Protocols: Simulating Your Scripts: In the Python Shell" from the API docs.

To simulate the script in an officially recommended way, we will first save our script above as .py file. We will name it "Test_script.py". Then, we will create another .py file in the same folder, and we will name it "simulate.py", and type in the following lines:

[34]:
```
from opentrons.simulate import simulate, format_runlog
protocol_filename = 'Test_script.py' # TODO: change this for a new file
protocol_file = open(protocol_filename)
runlog = simulate(protocol_file, file_name='')
print(format_runlog(runlog[0])) # Line modified compared to Opentrons Doc
```

```
C:\Users\User\.opentrons\deck_calibration.json not found. Loading defaults
C:\Users\User\.opentrons\robot_settings.json not found. Loading defaults

Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
        Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
        Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at
1.0 speed
        Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at
1.0 speed
        Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 150.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
        Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
        Aspirating 150.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at
1.0 speed
```

```
        Dispensing 150.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at
1.0 speed
        Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can see above that the log is nicely displayed. The building block commands within a complex command are indented for better readability. So why don't we create our script this way?

In fact, we could. However, it will make debugging much more complicated. So it is **not recommended**.

12.3 Exporting the log file for documentation

We can further expand the script above so that we do not just read the output from the console, but also save a text file as part of our lab notebook (works best if electronic).

[35]:
```
#%% Simulation

from opentrons.simulate import simulate, format_runlog
protocol_filename = 'Test_script.py' # TODO: change this for a new file
protocol_file = open(protocol_filename)
runlog = simulate(protocol_file, file_name='')
print(format_runlog(runlog[0])) # Line modified compared to Opentrons Doc

#%% Export simulated log file for documentation

simulated_log_filename = protocol_filename.split('.py')[0] + '_log.txt'
simulated_log = open(simulated_log_filename, 'w+', encoding = 'utf-8')
simulated_log.write(format_runlog(runlog[0]))
simulated_log.close()
```

```
Transferring 200.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
        Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
        Aspirating 200.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at
1.0 speed
        Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at
1.0 speed
        Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 150.0 from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
        Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
        Aspirating 150.0 uL from A1 of Corning 24 Well Plate 3.4 mL Flat on 2 at
1.0 speed
        Dispensing 150.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at
1.0 speed
        Dropping tip into A1 of Opentrons Fixed Trash on 12
```

Now we can go back to the same folder and open "Test_script_log.txt" and we can see that the output is exactly the same as what is shown on the console.

Advanced users can further add paths and folders so that the "simulate.py" file is placed inside a different folder.

We have now gone through the process for building a script and making it ready for execution!

13 (Part II) Customizing the liquid transfer process

Haivng a functional script is only the beginning. Very often we see unexpected behavior during the acutal pipetting steps and so many customizations are required before a protocol meets our demand. Fortuantely, the API provides multiple levels of control that we will find useful.

13.1 Customization on Transfers using Paramaters

The following section corresponds to "Complex Commands: Parameters" from the API docs.

The `pipette.transfer()` function can take in a number of keyword arguments to change details during the pipetting step. There are many . Since this depends, we will only go over the ones which are more frequently called.

[36]:
```
# Set up
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

r_pipette = protocol.load_instrument(
    'p300_single',
    'right',
    tip_racks=[tip_rack]
)
```

13.1.1 new_tip

By default the `pipette.transfer()` function sets `new_tip` to `'once'`.

[37]:
```
# Uses same tip for multiple transfer
r_pipette.transfer(200, plate1.wells()[0:2], plate1.wells()[2:4]
                   )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to C1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
```

```
Dispensing 200.0 uL into C1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 200.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into D1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

This is helpful for distributions but will cause contamination in many-to-many transfers. It is better to set `new_tip` to `'always'`.

[38]:
```
# Get a new tip every time
r_pipette.transfer(200, plate1.wells()[0:2], plate1.wells()[2:4],
                   new_tip='always'
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to C1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into C1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into D1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

This, however, can be an issue when you are pipetting large volumes.

[39]:
```
# This will get a new tip even if we are pipetting the same sample.
r_pipette.transfer(400, plate1.wells()[0], plate1.wells()[1],
                   new_tip='always'
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 400.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
```

```
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

13.1.2 mix_before / mix_after

Useful for protocols involving serial dilution. Note: check very carefully how it actually performs on the robot. Most of the time it does not do mixing properly.

[40]:
```
r_pipette.transfer(200, plate1.wells()[0], plate1.wells()[1],
                   mix_after=(3, 150) # mix for 3 cycles with volume of 150 µL
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from F1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Mixing 3 times with a volume of 150.0 ul
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

13.1.3 disposal_volume

If you need very accurate dispensing volume it is a good idea to aspirate an extra amount of volume so as to reduce residual volumes inside the tip. Note that the pipette discards the disposal volume at the trash though, so this will cause waste of the liquid sample.

[41]:
```
r_pipette.transfer(200, plate1.wells()[0], plate1.wells()[1],
                   disposal_volume=50
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from G1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Blowing out at A1 of Opentrons Fixed Trash on 12
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

13.2 Pipetting speed

It is possible to change the plunger flow rate. The corresponding API is given at "Pipettes: Plunger Flow Rates" from the API docs.

Changing plunger flow rate on the pipette is global and should affect all pipetting steps. In most cases we wish to slow down or speed up a few specific pipetting steps within a protocol, so if we change the plunger flow rate on the pipette, we will have to reset the speed to default before moving onto the next step. This is therefore not recommended to do so this way. Rather, we recommend using the "building block commands" (see a later section for the example) which has the optional argument of `rate` that allows us to modify plunger flow rate locally within a single step of aspiration or dispense.

13.3 Well location

The following section corresponds to "Labware: Specifying Position Within Wells" from the API docs.

Warning: read the warning in the docs on how collision would affect subsequent pipetteing

Some pipetting would benefit from aspirating or dispensing liquid above or slightly below the liquid surface, for example, aspirating supernatant from a centrifuged sample. The OT-2 does not sense the liquid level, but it is possible to instruct the pipette to aspirate/dispense at location relative to the top/bottom of a well.

This is done via calling an attribute of a well:

[42]:
```
plate1.wells_by_name()['A1'].top() # top of the well
```

[42]:
```
Location(point=Point(x=14.38, y=74.24, z=14.219999999999999), labware=A1 of
Corning 96 Well Plate 360 µL Flat on 1)
```

[43]:
```
plate1.wells_by_name()['A1'].top(-5) # 5 mm below the top of the well
```

[43]:
```
Location(point=Point(x=14.38, y=74.24, z=9.219999999999999), labware=A1 of
Corning 96 Well Plate 360 µL Flat on 1)
```

[44]:
```
plate1.wells_by_name()['A1'].bottom(10) # 10 mm above the bottom of the well
```

[44]:
```
Location(point=Point(x=14.38, y=74.24, z=13.549999999999999), labware=A1 of
Corning 96 Well Plate 360 µL Flat on 1)
```

Example:

[45]:
```
r_pipette.transfer(200, plate1.wells()[0],
                   plate1.wells()[1].top(-5) # location specified in a pipette.
 ↪transfer() complex command
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to B1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from H1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

Notice that the output command does not show that information in the console, but it will be carried out by the robot as instructed.

14 Achieving fine control through "building block" commands

The following section corresponds to "Building Block Commands" from the API docs.

It may appear that the `pipette.transfer()` function would suffice in many scenarios. In my personal experience, however, more often than not I would find unexpected behavior on the robot which jeopardize the entire experiment. For instance, mixing anything that is slighty more vicious than water, through capilliary action on the pipette tip, will lead to an incomplete dispense of liquid within each aspirate-dispense cycle. In the end, the tip used for transfer + mixing carried away a significant volume of liquid, and this could affect an experiment as simple as a serial dilution.

We will thus introduce the building block commands. Once we master them we will have much better control over the fine details of our pipetting steps. And this is in fact, surprisingly easy.

[46]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')
```

```
plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
plate2 = protocol.load_labware('corning_24_wellplate_3.4ml_flat', 2)
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 3)

r_pipette = protocol.load_instrument(
    'p300_single',
    'right',
    tip_racks=[tip_rack]
)
```

Consider the following `pipette.transfer()` function

[47]:
```
r_pipette.transfer(200, plate1.wells_by_name()['A1'], plate1.
 ↪wells_by_name()['A2'])
print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

If we rewrite the above by the building block commands:

[48]:
```
r_pipette.pick_up_tip() # no argument required since the robot tracks tip usage␣
 ↪for us
r_pipette.aspirate(200, # aspirate volume
                   plate1.wells_by_name()['A1'] # source well
                   )
r_pipette.dispense(volume=r_pipette.current_volume,
                    location=plate1.wells_by_name()['A2'] # destination well
                   )
r_pipette.drop_tip()

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

If we compare the outputs from `pipette.transfer()` and the building block commands, we can

see that there is virtually no difference in how the robot executes the function. The only difference is that the encapsulation line "Transferring… …" is no longer present.

Also note that we could have written `r_pipette.dispense(location=plate1.wells_by_name()['A2'])` directy instead.

14.1 Example 1: the blow_out=True issue

So why should we bother using building block commands? Let's consider the following scenairo

[49]:
```
r_pipette.transfer(200, plate1.wells_by_name()['A1'], plate1.
↪wells_by_name()['A2'],
                   blow_out = True # Note: key change here
                   )
print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Blowing out at A1 of Opentrons Fixed Trash on 12
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

As it turns out, Opentrons decided that, if you wish to expunge all liquid from the tip, those liquid will go to the trash instead of your destination well!

As seasoned pipette users we all know it isn't how we usually do things. Alas, the logic in liquid handling automation is that you should always include a disposal volume during pipetting to increase pipetting accuracy. There will always be residual liquid sticking to the interior walls of the tip and we cannot afford to manually inspect them while the robot is running. So yes, there is a reason behind that.

However, we sometimes work with precious samples and sometimes, we just have to sequeez out every bit of liquid from the tip into the well. Assuming we have to do so, we can rewrite the single `pipette.transfer()` function in the following way:

[50]:
```
r_pipette.pick_up_tip()
r_pipette.aspirate(200,
                   plate1.wells_by_name()['A1']
                   )
r_pipette.dispense(volume=r_pipette.current_volume,
                    location=plate1.wells_by_name()['A2']
                   )
r_pipette.blow_out(location=plate1.wells_by_name()['A2']) # if no argument,␣
↪blow out at current position
r_pipette.drop_tip()
```

```
print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Blowing out at A2 of Corning 96 Well Plate 360 µL Flat on 1
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

14.2 Example 2: Delay between aspiration and dispense

In another instance, say we wish to wait 5 seconds after aspiration, to allow a slighlty viscous liquid to be properly aspirated before dispensing it elsewhere.

[51]:
```
r_pipette.pick_up_tip()
r_pipette.aspirate(200, plate1.wells_by_name()['A1'])

protocol.delay(seconds=5) # additional line

r_pipette.dispense(location=plate1.wells_by_name()['A2'])
r_pipette.drop_tip()

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Delaying for 0 minutes and 5 seconds
Dispensing 200.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

14.3 Example 3: Mixing with customized speeds

One final example here is a highly customized mixing process for a peculiar need. Say, we wish to mix a sample 3 times, but: 1. The aspiration speed needs to be slower and the dispense speed needs to be faster. 2. We need to ensure all liquid is blown out before aspirating for the next cycle in mixing. 3. We want to dispense all liquid above the liquid level, not when the tip is still submerged inside the liquid.

This might look daunting, but with the building block commands it is in fact quite straightforward:

[52]:
```
well = plate1.wells_by_name()['A1']

r_pipette.pick_up_tip() # use the same tip for mixing
```

```
for i in range(3):

    r_pipette.aspirate(150, # mix volume= 150 µL
                       well,
                       rate = .75 # slower rate for aspiration
                      )

    r_pipette.dispense(location=well.top(-5), # 2 mm from top of the well,␣
 ↪ensures dispense above liquid level
                       rate = 1.5 # faster rate for dispense
                      )

r_pipette.drop_tip()

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from F1 of Opentrons 96 Tip Rack 300 µL on 3
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

It is also noteworthy that there is a `rate` argument that can modify pipetting speed, which is not applicable to the complex commands. The `rate` argument takes a float or integer coefficint and multiply the default speed (defined as rate=1) by that coefficient.

15 Writing reusable scripts

One advantage in using the API is that we could easily expand a script by copying and pasting its content and making small changes. That said, in programming it is better to keep codes as general as required to minimize efforts in reusing a script.

15.1 Labware and pipetting loading

For example, let's consider the scenarion that we are pipetting 200 µL of sample from one 96-well plate to another, say just from source well A1 to destination well A1, and we have to do it for 5 different destination plates. Using what we have written so far, we could write the following code:

[53]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

source_plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
dest_plate1 = protocol.load_labware('corning_96_wellplate_360ul_flat', 2)
dest_plate2 = protocol.load_labware('corning_96_wellplate_360ul_flat', 3)
dest_plate3 = protocol.load_labware('corning_96_wellplate_360ul_flat', 4)
dest_plate4 = protocol.load_labware('corning_96_wellplate_360ul_flat', 5)
dest_plate5 = protocol.load_labware('corning_96_wellplate_360ul_flat', 6)

tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', 7)

r_pipette = protocol.load_instrument(
    'p300_single',
    'right',
    tip_racks=[tip_rack]
)

dest_plates = [dest_plate1, dest_plate2, dest_plate3, dest_plate4, dest_plate5]
for dest_plate in dest_plates:
    r_pipette.transfer(200,
                       source_plate.wells_by_name()['A1'],
                       dest_plate.wells_by_name()['A1']
                      )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 2
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 3
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 3 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 4
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 7
```

```
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 4 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 5
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 5 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 6
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 6 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

However, we can easily imagine that this is unsustainable as we further scale up the experiment, or if we start a new experiment using different labware.

We can restructure the code above to simplify labware loading and improve readability and reusability:

[54]:
```python
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

#%% Deck layout and pipette setting
deck_layout = {
    1:'corning_96_wellplate_360ul_flat',
    2:'corning_96_wellplate_360ul_flat',
    3:'corning_96_wellplate_360ul_flat',
    4:'corning_96_wellplate_360ul_flat',
    5:'corning_96_wellplate_360ul_flat',
    6:'corning_96_wellplate_360ul_flat'
}

l_pipette_name = ''
r_pipette_name = 'p300_single'

l_tip_rack_slots = []
l_tiprack_name = ''

r_tip_rack_slots = [7]
```

```
r_tiprack_name = 'opentrons_96_tiprack_300ul'

#%% Deck and instrument loading
# DO NOT EDIT

dlw = {} # dlw stands for deck_labware
for slot, labware_item in deck_layout.items():
    dlw.update({slot:protocol.load_labware(labware_item, slot)})

if l_tip_rack_slots:
    l_tip_racks = [protocol.load_labware(l_tiprack_name, slot) for slot in␣
 ↪l_tip_rack_slots]

if r_tip_rack_slots:
    r_tip_racks = [protocol.load_labware(r_tiprack_name, slot) for slot in␣
 ↪r_tip_rack_slots]

if l_pipette_name:
    l_pipette = protocol.load_instrument(r_pipette_name, mount = 'left',␣
 ↪tip_racks = l_tip_racks)

if r_pipette_name:
    r_pipette = protocol.load_instrument(r_pipette_name, mount = 'right',␣
 ↪tip_racks = r_tip_racks)

#%% Commands

for dest_slot in range(1, 6):
    r_pipette.transfer(200,
                       dlw[1].wells_by_name()['A1'],
                       dlw[dest_slot].wells_by_name()['A1']
                      )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 2
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 7
```

```
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 2 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 3
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 3 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 4
Picking up tip from D1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 4 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 200.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A1 of
Corning 96 Well Plate 360 µL Flat on 5
Picking up tip from E1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 200.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 200.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 5 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can see that dictionary `deck_layout` now lists all the labware containers except tip racks that sit on the deck. It is easier to keep track of and modify labware items this way. The tip racks slots are seperated in another list variable because they need to associate with the pipette. By doing so, calling a labware item now done via `dlw[slot_number]` which is useful because labware items can now be looped by their slot numbers.

Note that this is only a personal recommendation. In fact, I usually prefer commenting lines of unused pipettes out to avoid debugging problems. The key in structuring the script is to separate codes into *constant* and *variable* regions (just like an antibody) so as to minimize effort in tracking what needs to be changed every time.

15.2 Reusable pipetting functions

We have covered building block commands. If such building block commands are repeated over and over again it would be advantageous to wrap them inside function. We will reuse the example above on mixing.

[55]:
```
# The original script
well = dlw[1].wells()[0]
```

```
r_pipette.pick_up_tip() # use the same tip for mixing

for i in range(3):

    r_pipette.aspirate(150, # mix volume= 150 µL
                       well,
                       rate = .75 # slower rate for aspiration
                      )

    r_pipette.dispense(location=well.top(-5), # 2 mm from top of the well,␣
 ↪ensures dispense above liquid level
                       rate = 1.5 # faster rate for dispense
                      )

r_pipette.drop_tip()

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from F1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

[56]:
```
# A generalized function for mixing in a particular way

def custom_mix(pipette, mix_vol, well):
    pipette.pick_up_tip() # use the same tip for mixing

    for i in range(3):
        pipette.aspirate(mix_vol, # mix volume= 150 µL
                         well,
                         rate = .75 # slower rate for aspiration
                        )
```

```
        pipette.dispense(location=well.top(-5), # 2 mm from top of the well,␣
↪ensures dispense above liquid level
                         rate = 1.5 # faster rate for dispense
                        )

    r_pipette.drop_tip()

for i in range(2):
    custom_mix(r_pipette, 150, dlw[1].wells()[i])

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Picking up tip from G1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Picking up tip from H1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Aspirating 150.0 uL from B1 of Corning 96 Well Plate 360 µL Flat on 1 at 0.75
speed
Dispensing 150.0 uL into B1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.5
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

Notice that we have called the `custmom_mix()` function twiced using a for-loop. This would be more readable than having multiple layers of nested for-loops. Also, the object `pipette` is being used as an argument in the custom function, instead of being called directly for its own command.

Note: In the final script to be executed, in theory these custom function can be placed inside or outside the `run()` function. Both should work, however, for beginners it is recommended that

these functions be placed **after** labware items are loaded, since it is easy to miss variables that are defined outside the function.

16 Manual intergration with Excel

Many biologists use Microsoft Excel for data analysis and for planning experiments. We introduce below one of the ways to harness the user-friendly tabular layout of Excel to generate Python parsable commands for the OT-2.

First, we will create an Excel file that look like the following:

	A	B	C	D	E	F
1	volume	source_plate	source_well	dest_plate	dest_well	command
2	5	1	A1	3	D4	
3	10	2	B2	4	F5	
4	15	4	C3	5	G8	

Then, we will starting typing, under the command column, in cell `F1`, we will type:

```
=CONCAT("'",B3, "$", C3, "_", D3, "->", E3, "_", F3,"',")
```

and then we will autofill the rest of the `command` column using the same formula.

This will give us a table that looks like the following:

	A	B	C	D	E	F
1	volume	source_plate	source_well	dest_plate	dest_well	command
2	5	1	A1	3	D4	'5$1_A1->3_D4',
3	10	2	B2	4	F5	'10$2_B2->4_F5',
4	15	4	C3	5	G8	'15$4_C3->5_G8',

Reading this table above, we can see that we have condensed the minimal information for a pipetting step into single machine line, which remains readable by human, in the form of

volume $ source slot _ source well -> dest slot _ dest well

Each line can be presented as a string to Python within a Python list. We can then write a corresponding Python section to decode the information.

[57]:
```python
# This cell is for preparation only

from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

#%% Deck layout and pipette setting
deck_layout = {
    1:'corning_96_wellplate_360ul_flat',
    2:'corning_96_wellplate_360ul_flat',
```

```
    3:'corning_96_wellplate_360ul_flat',
    4:'corning_96_wellplate_360ul_flat',
    5:'corning_96_wellplate_360ul_flat'
}

l_pipette_name = ''
r_pipette_name = 'p300_single'

l_tip_rack_slots = []
l_tiprack_name = ''

r_tip_rack_slots = [7]
r_tiprack_name = 'opentrons_96_tiprack_300ul'

#%% Deck and instrument loading
# DO NOT EDIT

dlw = {} # dlw stands for deck_labware
for slot, labware_item in deck_layout.items():
    dlw.update({slot:protocol.load_labware(labware_item, slot)})

if l_tip_rack_slots:
    l_tip_racks = [protocol.load_labware(l_tiprack_name, slot) for slot in␣
 ↪l_tip_rack_slots]

if r_tip_rack_slots:
    r_tip_racks = [protocol.load_labware(r_tiprack_name, slot) for slot in␣
 ↪r_tip_rack_slots]

if l_pipette_name:
    l_pipette = protocol.load_instrument(r_pipette_name, mount = 'left',␣
 ↪tip_racks = l_tip_racks)

if r_pipette_name:
    r_pipette = protocol.load_instrument(r_pipette_name, mount = 'right',␣
 ↪tip_racks = r_tip_racks)
```

[58]:
```
#%% Transfer information

transfer_commands = [

    # The following contents are copied and pasted from the Excel spreadsheet
    '5$1_A1->3_D4',
    '10$2_B2->4_F5',
    '15$4_C3->5_G8',

]
```

```
#%% Decoding the information and executing the commands

for command in transfer_commands:

    volume = int(command.split('$')[0]) # freshnly split volume is a str, need␣
↪to convert to int

    transfer_path = command.split('$')[1] # breaks down the first command to␣
↪'1_A1->3_D4'

    source = transfer_path.split('->')[0] # further breaks down the first␣
↪"transfer_path" to '1_A1'
    dest = transfer_path.split('->')[1] # further breaks down the first␣
↪"transfer_path" to '3_D4'

    source_slot, source_well = source.split('_') # double assignment,␣
↪source_slot='1' and source_well='A1'
    dest_slot, dest_well = dest.split('_')

    # We used int for slots in "deck_layout", so we need to convert the type too
    source_slot = int(source_slot)
    dest_slot = int(dest_slot)

    # Commands
    r_pipette.transfer(volume,
                       dlw[source_slot].wells_by_name()[source_well],
                       dlw[dest_slot].wells_by_name()[dest_well]
                      )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 5.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to D4 of
Corning 96 Well Plate 360 µL Flat on 3
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 5.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0 speed
Dispensing 5.0 uL into D4 of Corning 96 Well Plate 360 µL Flat on 3 at 1.0 speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 10.0 from B2 of Corning 96 Well Plate 360 µL Flat on 2 to F5 of
Corning 96 Well Plate 360 µL Flat on 4
Picking up tip from B1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 10.0 uL from B2 of Corning 96 Well Plate 360 µL Flat on 2 at 1.0
speed
Dispensing 10.0 uL into F5 of Corning 96 Well Plate 360 µL Flat on 4 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

```
Transferring 15.0 from C3 of Corning 96 Well Plate 360 µL Flat on 4 to G8 of
Corning 96 Well Plate 360 µL Flat on 5
Picking up tip from C1 of Opentrons 96 Tip Rack 300 µL on 7
Aspirating 15.0 uL from C3 of Corning 96 Well Plate 360 µL Flat on 4 at 1.0
speed
Dispensing 15.0 uL into G8 of Corning 96 Well Plate 360 µL Flat on 5 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can further simplify the script, but the above version is a sufficiently good start to allow us to make good use of Excel to assist our experiment planning on an OT-2. The choices of delimiting characters can also be altered depeneding on our own need.

It is possible to do .csv import in Python but it is not recommended, see Additional Remarks 2.

17 Additional remarks

1. It is faster to use `pipette.transfer()` with list of well objects than looping a a single `pipette.transfer()` function multiple times. The difference in speed in unnoticeable on a PC but could mean 4-5 minutes of wait when API of a long protocol is compiled on a Pi-powered robot.

The following example illustrates a comparison of the two methods, both trying to pipette 200 µL from well A1 to A7, B1 to B7, … H6 to H12, for a total of 48 transfer steps.

A single complex function processes multi-to-multi transfers much more efficiently, at the expense of losing fine-tuned control at the atomic movement level.

[59]:
```
# Set up
import time
from opentrons import simulate
protocol = simulate.get_protocol_api('2.2')
tip_rack = protocol.load_labware('opentrons_96_tiprack_300ul', location = '1')
plate = protocol.load_labware('corning_96_wellplate_360ul_flat', location = '2')
pipette = protocol.load_instrument('p300_single', mount = 'right', tip_racks =␣
↪[tip_rack])
print('Run time comparison:')

# Method 1: List for looping pipette.transfer
transfer_info = {x: x + 48 for x in range(0,48)}

start_time = time.time()
for source_index, dest_index in transfer_info.items():
    pipette.transfer(200, plate.wells(source_index), plate.wells(dest_index))
end_time = time.time()
time_diff = end_time - start_time
print('Loop method: ' + str(time_diff))

# Method 2: Looping through a series of wells
```

```
source_wells = plate.columns()[0:3]
dest_wells = plate.columns()[4:7]
start_time = time.time()
pipette.transfer(200, source_wells, dest_wells)
end_time = time.time()
time_diff = end_time - start_time
print('Well object method: ' + str(time_diff))
```

```
Run time comparison:
Loop method: 0.843747615814209
Well object method: 0.1515941619873047
```

This difference looks insignificant to us, but we should remember that our script is executed on a Raspberry Pi inside the OT-2 and it will be a difference of minutes if our script gets longer and longer.

2. It is possible to import csv files into our Python script. However, that requires a manual upload of the csv file to the storage space through the OT-2's Jupyter Notebook terminal. Note that this is **not** the Jupyter Notebook of the computer that connects to the OT-2 but the Notebook that is executed from the Raspberry Pi inside the OT-2. Further instructions for parsing the csv files are also provided by Opentrons.

We strongly **discourage** doing so because it entails lab management problems. Labs with 10+ years of history often see their PCR machines storing many outdated protocols from alumni and eventually run of space. The Raspberry Pi inside the OT-2 has limited storage, and new members who do not wish to risk offending anyone will allow those old protocols/files to sit there forever and this will slowly chip away space for future OT-2 firmware upgrades. In contrast, the robot should only store the most recently uploaded protocol. Therefore, encoding all information inside an executable script is preferred.

3. Labware objects issues: It is not important to have the exact labware name for a labware that we do not have. For example, our lab uses 96-well plates from Greiner Bio-One but we just use "corning_96_wellplate_360ul_flat" as the labware name and it works just fine. That is not always the case, however, especially when it comes to 96-well PCR plates. There are considerable differences in well heights across different manufacturers and so this needs to be tested case-by-case.

4. When accessing wells on a labware, the direct access of wells without using a method (i.e., `plate['A1']`) is strongly discouraged. It might make it easier to read and save time in writing but when we start to build complex functions, it makes debugging much more difficult.

5. It is possible to share tip rack between two pipettes and tip tracking will be fine if you are using single channel pipettes. However, that is a bad habit and **should be discouraged**. Consider the following scenario where we use a 8-channel pipette and a single channel pipette together.

[60]:
```
from opentrons import simulate
protocol = simulate.get_protocol_api(version = '2.4')

#%% Deck layout and pipette setting
```

```
deck_layout = {
    1:'corning_96_wellplate_360ul_flat'
}

l_pipette_name = 'p300_single'
r_pipette_name = 'p300_multi'

shared_tip_rack_slots = [2]
shared_tiprack_name = 'opentrons_96_tiprack_300ul'

dlw = {} # dlw stands for deck_labware
for slot, labware_item in deck_layout.items():
    dlw.update({slot:protocol.load_labware(labware_item, slot)})

if shared_tip_rack_slots:
    shared_tip_racks = [protocol.load_labware(shared_tiprack_name, slot) for␣
 ↪slot in shared_tip_rack_slots]

if l_pipette_name:
    l_pipette = protocol.load_instrument(r_pipette_name, mount = 'left',␣
 ↪tip_racks = shared_tip_racks)

if r_pipette_name:
    r_pipette = protocol.load_instrument(r_pipette_name, mount = 'right',␣
 ↪tip_racks = shared_tip_racks)

# Commands
l_pipette.transfer(volume,
                   dlw[1].wells_by_name()['A1'],
                   dlw[1].wells_by_name()['A2']
                  )

r_pipette.transfer(volume,
                   dlw[1].wells_by_name()['A1'],
                   dlw[1].wells_by_name()['A2']
                  )

print(*protocol.commands(), sep = '\n')
protocol.clear_commands()
```

```
Transferring 15.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A1 of Opentrons 96 Tip Rack 300 µL on 2
Aspirating 15.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 15.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
```

```
Dropping tip into A1 of Opentrons Fixed Trash on 12
Transferring 15.0 from A1 of Corning 96 Well Plate 360 µL Flat on 1 to A2 of
Corning 96 Well Plate 360 µL Flat on 1
Picking up tip from A2 of Opentrons 96 Tip Rack 300 µL on 2
Aspirating 15.0 uL from A1 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dispensing 15.0 uL into A2 of Corning 96 Well Plate 360 µL Flat on 1 at 1.0
speed
Dropping tip into A1 of Opentrons Fixed Trash on 12
```

We can see that left P300S took a tip from A1 in the tip rack but then the right P300M took tips from B1-H1 + (missing tip). At this point the API is not smart enough to tell how to pick up tips. This can be overcome using `pipette.pick_up_tip()` but the risk is not worth the tips being saved. It is better to always **assign separate tip racks** to the two pipettes.