RobotEvo Documentation

Release 2.10.00

Ariel Vina-Rodriguez

Dec 15, 2019

Contents:

RobotEvo generates Freedom EVOware scripts for the Freedom EVO universal pipetting robot from TECAN. It provides new layers of abstraction to offer a higher level programing model that allows a more direct programing of the steps needed in a typical biochemical/biological pipetting protocol like RNA extraction.

Why RobotEvo?

(adapted from my PhD Thesis: RNA virus detection and identification using techniques based on DNA hybridization)

- **Programming automation of RNA extraction:**

Usually, prior to proceed to the application of the DNA-hybridization-based technique, like RT-qPCR, the viral RNA need to be extracted. We used well established methods and commercially available kits based on columns (RNeasy Mini Kit or QIAamp Viral RNA Mini Kit, QIAGEN, Hilden, Germany) or magnetized particles (NucleoMag® VET kit) from MACHEREY-NAGEL, Durel, Germany) to achieve the separation, either automatically, using pipetting robots, or manually.

Especially useful was the combination of magnetized particles with a Freedom EVO universal pipetting robot from TECAN, Mannerdorf, Switzerland.

Using the provided software (Freedom EVOware) it was comfortable to write simple and specific pipetting protocols in a semi visual way. Unfortunately, writing more complex or flexible protocols (for example to accommodate arbitrary number of samples or minor modifications of the protocols) was very time consuming and error prone. You are compeled to use variables and program-control-flow structures like IF and LOOP. But you will find that there is a poor or no support of variables of different types, arrays, structural-programing and objects within the provided scripting language.

More important, the powerfull validation and visualization tools provided by the script editor are full sopprted only in relatively lineal and simple scripts, considering only the "default" values of the variables, and thus, the default flow-path of the program, not detecting problems in the alternative paths, likely to be found in most runs.

To overcome these limitations and afford automation, a new software was written in Python, "RobotEvo", which generates the scripts for the robot. This new Python library provide new layers of abstraction to offer a higher level programing model to allow a more direct programing of the steps needed in a typical biochemical/biological pipetting protocol like RNA extraction. The layers of the implementation are: a parser and a generator (module :file: **'instructions.py'_** of the "low-level" instruction set directly usable by the provided Freedom EVOware software; a set of "modes" to provide the desired kind of output (human readable comments, separated instructions, EVOware scripts, etc., in module :file: **'evo_mode.py'_**; a model of the state of the robot to detect possible errors prior to the generation of the script by tracking what volume of what mix of reagents contains at each moment each reservoir or tip (module [Robot](https://github.com/qPCR4vir/robotevo/blob/master/EvoScriPy/Robot.py) – this is a **_novel_** functionality impossible to achieve with the original software); low level pipetting instructions (like aspirate a specific

liquid volume from a given vial into a tip); a higher level command set (like distribute some reagent into each sample, in module [protocol steps](https://github.com/qPCR4vir/robotevo/blob/master/EvoScriPy/protocol_steps.py)) to directly program high-level, more realistic protocol scripts including a base template for a full protocol; and, finally, a set of facilities to declare the reagents (module Reagent) and the labwares (like reaction tubes, racks of tubes, racks of tips, cuvettes, etc. in module [Labware](https://github.com/qPCR4vir/robotevo/blob/master/EvoScriPy/Labware.py)).

For the protocol for RNA extraction (module RNAextractionMN_Mag_Vet) the set of used labwares and reagents are declared first. Immediately an automatically generated check-list is presented to the human operator (a graphic user interface – from module GUI). After a possible adjustment of the predefined parameters (without any programming) the program go through a few high-level-defined protocol steps of distributeing buffers, mixing, washing, incubating, etc. generating a very detailed set of low-level instructions for the physical robot in a script to be imported and visualized in the TECAN Freedom EVOware software. The obtained script is very long but structurally very simple and well commented, which facilitates the visual control of each instruction prior to real pipetting.

- modules.jpg

Users of RobotEvo

To help understand what are the expected user cases we will classify potential users in categories. For each we will then try to describe needed skills, adventages of using RobotEvo, basic usage and tips.

2.1 Category A user: the real-robot operator

This is the basic user for which RobotEvo was writen and for which all the other Categories-Users work.

Expected skills: no special skills expected, In fact, for basic, standard usage they don't need to know about RobotEvo existence. Just normal, basic usage of the original Evoware software.

2.2 Category B user: basic RobotEvo/python operator

Use RobotEvo to generate evoware scripts and load it into evoware, saving correct scripts where they will be accessed by Category A users.

Expected skills: run python scripts; locate and copy files to the location from were User A can load to the EVOware editor.

2.3 Category C user: protocol adaptor

Modify scrips, uses the autogenerated RobotEvo protocol GUI or existing scripts to modify the imput ranges, variants and some other options in programmed protocols, or making minor modifications to them;

Expected skills: good understanding of the affected protocol.

2.4 Category D user: protocol developer

Developer of protocols.

Expected skills:

- python;

- RobotEvo APIs;

- deep understanding of the implemented (bio)chemical protocols.

2.5 Category E user: RobotEvo developer

Core developer of RobotEvo.

Expected skills:

- python;

- deeper understanding of RobotEvo implementation;

- know the original API of the programmed robot;

- understanding of the implemented protocols;

- testing;

- git-GitHub.

- How to run an existing protocol?

- How does it works?

- A Hello World! example.

- How to modify an existing protocol?

- Now to write a new protocol?

How to run an existing protocol?

Make sure you have a working python3 interprete in your device (PC, tablet, smartphone, etc.) and a copy of RobotEvo (downloaded or cloned from GitHub).

Now, the simplest way is to run the python script containing the protocol, providing it have a "main" function.

For better control, in a your onw python script, import the desired protocol and just create an instance (python object) of the protocol, possibly setting some of the constructor parameters, and call the *.run()* method of that object. You can see many examples of this usage in the script robotevo/protocols/test.py This will create a set of files with the generated evoware scripts, a human readable protocol, and comments, possibly including warnings. It may abort with more or less detailed messages about the errors.

Alternatively run *GUI.py* (only in devices with a functioning standard python module "tkinter") to select the protocol, the desired protocol "variant" and change some other minor parameters like number of samples or required reagents volume.

To make the actual pippeting in the real robot, open the generated *.esc* script in the EVOware editor. It will alert you that the check sum have not been set, which in this case just flags the fact that this is a newly generated script you have not run yet. Accept to load it in the EVOware script editor. Here you will have very good assistance to visualize the details of each step and to do a normal, full TECAN validation of the correctness of the script.

Use the information from the visual worktable map to physically setup the labware. Use the detailed comments automatically inserted by RobotEvo in the script or the associated *.protocol.txt* file to fill the expected initial volume of each reagent.

Use EVOware to run the script as usually.

3.1 How does it works?

Already the creation of the protocol object will run some "boilerplate" code to setup things we need to run the useful part of our protocol.

For example it will parse the provided worktable template file (a *.ewt* or just a compatible *.esc* evoware file) and will remember (in a sort of *map*) all the labware present in the worktable, including its unique name, type and location.

It will also initialize some other characteristics of the used robot (not present in the worktable file) like number of tips in the LiHa, etc.

Additionally it will set the desired EvoMode: what kind of output we want to produce - normally an evoware script (his generated script will include again all the information for the worktable), but also a human readable protocol, etc.

By running *.run()* we "create" or "get", from the parsed worktable file, labwares, like multiplates, tube racks, etc, and "create" the reagents defined there in the script, including location in the worktable, volume, etc. This make possible for the "internal iRobot" to model or track the content of each well, and to detect (and report) potential logical errors in the protocol.

If at this point the protocol include a call to *.cehcklist()* instructions will be generated to inform to the human robot-operator **at run time**, the positions and initial quantity of all reagents he need to make sure are in place. If a GUI is in use and was previously created a new sub-GUI will be automatically generated to show all the details of the defined reagents making possible to change some properties without modifying programmatically the protocol.

A typical protocol will use the high level instructions inherited from *protocol_steps*, like *transfer*, *distribute*, *with tips*, etc., to express the "physical" protocol. This instructions are in turn internally implemented using lower level instructions like *aspirate*, *get tips*. etc. Each of this low level intructions will interact with the selected EvoMode to generate the corresponding instructions in the EVOware script and to check errors and change the state of the internally modeled *iRobot*, including the liquid volume in each well and tip and many other details.

3.2 A Hello World! example.

Let create the classical, in the the world of programming, Hello World! example. It will just shows that message in the screen of the PC controlling the robot and will wait for user confirmation producing a typical sound.

By running the script:

```python
#
"""
Hello World!
------------

The classical, in the the world of programming, Hello World! example.
It will just shows that message in the screen of the PC controlling the robot and
→will wait
for user confirmation producing a typical sound.

"""

from EvoScriPy.protocol_steps import *


class HelloWorld(Protocol):
    """
    This is a very general protocol.
    Normally you will inherit from a Protocol class adapted to your real robot.
    """
    name = "Hello World"

    def __init__(self, GUI                         = None,
                       output_filename             = None,
                       worktable_template_filename = None):
        this = Path(__file__).parent
        Protocol.__init__(self,
```

```
                                GUI                           = GUI,
                                output_filename               = output_filename or this /
→'scripts' / 'hello_world',
                                worktable_template_filename   = worktable_template_filename␣
→or this / 'hello_world.ewt')

    def run(self):
        self.check_list()
        self.user_prompt("Hello World!")
        self.done()


if __name__ == "__main__":
    HelloWorld().run()
```

[IMPORTANT: replace the *worktable_template_filename* argument with any valid -for your very onw robot- worktable template (*.ewt*) or script (*.esc*).]

we will have some files (currently 4) generated with names following the pattern of the *output_filename* constructor argument: in particular '../current/tests/hello_world.esc' will contain a new evoware script you can load into the Freedom evoware editor. After you agree to use the script with an still unvalidated check-summe you will see it just contain an instruction for a simple user promt. By using evoware to run this script you will get:
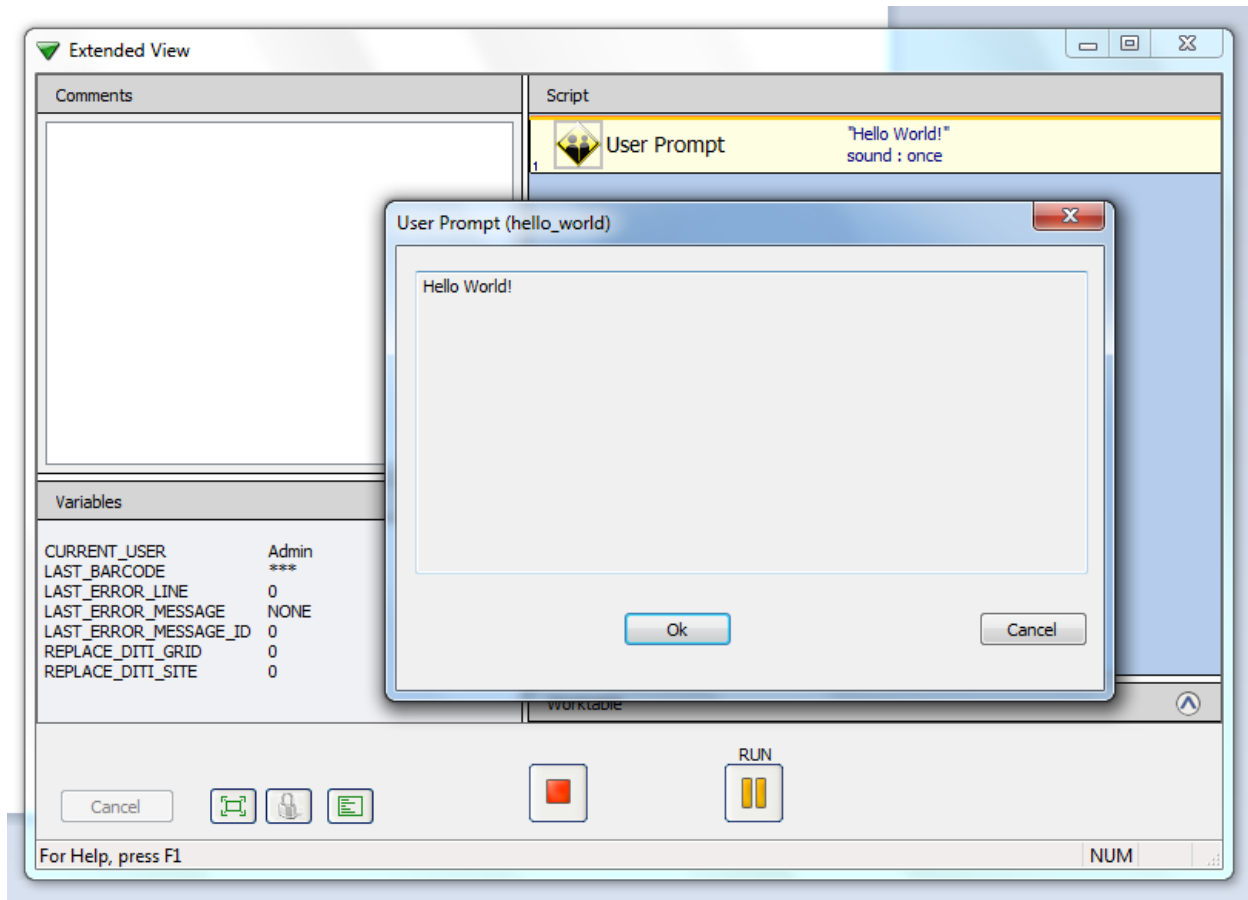
How to modify an existing protocol?

How to write a new protocol?

```python
# author Ariel Vina-Rodriguez (qPCR4vir)
# this example is free to use at your onw risk
# 2019-2019
"""
Prepare serial dilutions of two mixes.
-------------------------------------

"""

__author__ = 'Ariel'

# Tutorial

from protocols.evo200_f.evo200_f import *


class DemoTwoMixes(Evo200_FLI):
    """
    Prepare two 1:10 serial dilutions of two different mixes each in 'n' 100 uL final
↪volume wells
    (each in a microplate, the second one to be moved to the working position).

    'mix1' and 'mix2' are diluted separately in n wells 1:10 (mix1_10 and mix2_10
↪respectively) using
    a provided "buffer". From those wells a portion is transferred to the final 1:100
↪dilutions
    (mix1_100 and mix2_100 respectively) to fv=100 uL final volume

    One way to achieve this:
    - Calculate how much to transfer from each mix1_10 to mix1_100. v_mix1_10_100= fv/
↪10 and from diluent.
    - Calculate how much to distribute from mix1 to each mix1_10 and from diluent.
    - Define a reagent `mix1` and `mix2`in an Eppendorf rack (labware) for the
↪calculated volume per "sample" (mix1_10 or 2).
```

```
    - Define a reagent `buffer` in a 100 mL cubette `BufferCub` for the total volume␣
↪per "sample".
    - Generate check list
    - Transfer plate 2 from the original location `plate2` to the final location␣
↪`plate2-moved`
    - Define derived reagents for diluted mixes
    - Distribute mix1 and buffer into mix1_10 and similar with mix2
    - Transfer from mix1_10 to mix1_100 and distribute buffer here. The same with␣
↪mix2_10
    """

    name = "Prefill one plate with Buffer."
    min_s, max_s = 1, 96/2

    # for now just ignore the variants
    def def_versions(self):
        self.versions = {'No version': self.V_def}

    def V_def(self):
        pass

    def __init__(self,
                 GUI                         = None,
                 num_of_samples: int         = None,
                 worktable_template_filename = None,
                 output_filename             = None,
                 first_tip                   = None,
                 run_name: str               = ""):

        this = Path(__file__).parent

        Evo200_FLI.__init__(self,
                            GUI                         = GUI,
                            num_of_samples              = num_of_samples or␣
↪DemoTwoMixes.max_s,
                            worktable_template_filename = worktable_template_filename␣
↪or
                                                          this / 'demo-two.mixes.
↪Evo200example.ewt',
                            output_filename             = output_filename or this /
↪'scripts' / 'two.mixes',
                            first_tip                   = first_tip,
                            run_name                    = run_name)

    def run(self):
        self.initialize()                                       # set_EvoMode␣
↪and set_defaults() from Evo200

        self.check_initial_liquid_level = True
        self.show_runtime_check_list    = True

        num_of_samples = self.num_of_samples
        assert self.min_s <= num_of_samples <= self.max_s, "In this demo we want to␣
↪set 2x num_of_samples in a 96 well plate."
        wt             = self.worktable

        self.comment('Prefill a plate with some dilutions of two master mix and␣
↪Buffer Reagent for {:d} samples.'
```

```
                    .format(num_of_samples))

        buf_cuvette   = wt.get_labware("BufferCub", labware.Trough_100ml)      # Get
→Labwares from the work table
        master_mixes_ = wt.get_labware("mixes", labware.Eppendorfrack)

        buf_per_sample =0
        fv = 100

        v_mix_10_100 = fv / 10                                      # to be transferred
→from mix1_10 to mix1_100
        buf_mix_100 = fv - v_mix_10_100
        buf_per_sample += buf_mix_100

        v_mix_10 = (fv + v_mix_10_100)/10                           # to be distribute from
→original mix1 to mix1_10
        buf_mix_10 = (fv + v_mix_10_100) - v_mix_10
        buf_per_sample += buf_mix_10

        # Define the reagents in each labware (Cuvette, eppys, etc.)

        buffer = Reagent("Buffer ", buf_cuvette, volpersample  = buf_per_sample,
                                                 def_liq_class  = self.Water_wet,
                                                 num_of_samples = 2 * self.num_of_
→samples)

        mix1 = Reagent("mix1", master_mixes_, volpersample  = v_mix_10,
                                              def_liq_class  = self.Water_wet,
                                              num_of_samples = self.num_of_samples)

        mix2 = Reagent("mix2", master_mixes_, volpersample  = v_mix_10,
                                              def_liq_class  = self.Water_wet,
                                              num_of_samples = self.num_of_samples)

        # Show the check_list GUI to the user for possible small changes

        self.check_list()

        instructions.wash_tips(wasteVol=5, FastWash=True).exec()

        plate1 = wt.get_labware("plate1", '96 Well Microplate')
        plate2 = wt.get_labware("plate2", '96 Well Microplate')

        new_location = wt.get_labware("plate2-moved").location

        Reagent.use_minimal_number_of_aliquots = False           # Define derived
→reagents  --------------------

        mix1_10 = Reagent(f"mix1, diluted 1:10",
                      plate1,
                      initial_vol = 0.0,
                      num_of_aliquots= num_of_samples,
                      def_liq_class = self.Water_free,
                      excess       = 0)

        mix2_10 = Reagent(f"mix2, diluted 1:10",
                      plate2,
```

```python
                      initial_vol = 0.0,
                      num_of_aliquots= num_of_samples,
                      def_liq_class = self.Water_free,
                      excess       = 0)

    mix1_100 = Reagent(f"mix1, diluted 1:100",
                      plate1,
                      wells       = 'A07',
                      initial_vol = 0.0,
                      num_of_aliquots= num_of_samples,
                      def_liq_class = self.Water_free,
                      excess       = 0)

    mix2_100 = Reagent(f"mix2, diluted 1:100",
                      plate2,
                      wells       = 'A07',
                      initial_vol = 0.0,
                      num_of_aliquots= num_of_samples,
                      def_liq_class = self.Water_free,
                      excess       = 0)

    instructions.transfer_rack(plate2, new_location).exec()                    #␣
↪just showing how RoMa works.

    with group("Fill plate with mixes "):

        self.user_prompt("Put the plates for Buffer ")

        with self.tips(reuse=True, drop=False):
            self.distribute(reagent          = mix1,
                            to_labware_region = mix1_10.select_all())

        with self.tips(reuse=True, drop=False):
            self.distribute(reagent          = mix2,
                            to_labware_region = mix2_10.select_all())

        with self.tips(reuse=True, drop=False):
            self.distribute(reagent=buffer, to_labware_region=mix1_10.select_
↪all(), volume=buf_mix_10)
            self.distribute(reagent=buffer, to_labware_region=mix2_10.select_
↪all(), volume=buf_mix_10)

        with self.tips(reuse=True, drop=False):
            wells_100 = mix1_100.select_all()
            self.transfer(from_labware_region = mix1_10.select_all(),
                          to_labware_region   = wells_100,
                          volume              = v_mix_10_100)

        with self.tips(reuse=True, drop=False):
            wells_100 = mix2_100.select_all()
            self.transfer(from_labware_region = mix2_10.select_all(),
                          to_labware_region   = wells_100,
                          volume              = v_mix_10_100)

        with self.tips(reuse=True, drop=False):
            self.distribute(reagent=buffer, to_labware_region=mix1_100.select_
↪all(), volume=buf_mix_100)
```

```python
            self.distribute(reagent=buffer, to_labware_region=mix2_100.select_
→all(), volume=buf_mix_100)

            self.drop_tips()

        self.done()


if __name__ == "__main__":
    p = DemoTwoMixes(num_of_samples= 4,
                     run_name= "_4s_mix_1_2")

    p.use_version('No version')
    # p.set_first_tip('A01')
    p.run()
```

we will have something like:

1	Comment	Prefill a plate with some dilutions of two master mix and Buffer Reagent for 4 samples.
2	Set DiTi position	DiTi 1000ul Grid : 55, Site : 1, Position in labware : 1
3	User Prompt	"Check Buffer in ['well 1 in BufferCub : with 1497.0 uL of Buffer ']☐☐Check mix1 in ['well 1 in mixes : with 41.0 uL of mix1']☐☐Check mix2 in ['well 2 in m sound : once
4	Comment	Check: ['well 4 in BufferCub : with 1497.0 uL of Buffer ']
5	Get DiTis	DiTi 1000ul
6	Detect Liquid	Water free "BufferCub" (Col. 1, Row 4)
7	Drop DiTis	"DiTi Waste"
8	User Prompt	"" sound : once
9	Comment	Check: ['well 1 in mixes : with 41.0 uL of mix1']
10	Get DiTis	DiTi 1000ul
11	Detect Liquid	Water free "mixes" (Col. 1, Row 1)
12	Drop DiTis	"DiTi Waste"
13	User Prompt	"" sound : once

13	User Prompt	"" sound : once	
14	Comment	Check: ['well 2 in mixes : with 41.0 uL of mix2']	
15	Get DiTis	DiTi 1000ul	
16	Detect Liquid	Water free "mixes" (Col. 1, Row 2)	
17	Drop DiTis	"DiTi Waste"	
18	User Prompt	"" sound : once	
13	Wash Tips	5 + 10 ml	
20	Transfer Labware	Source: Grid '55,' Site '3'; Destination: Grid '55', Site '2'; Narrow (ROMA 1); Move to home position	
21	Group	Fill plate with mixes	
22	User Prompt	"Put the plates for Buffer " sound : once	
23	Group	Spread: 10.0 µL of mix1	
24	Comment	Spread: 10.0 µL of mix1 (41.6 µL total) from [grid:12 site:1 ['well 1 in mixes : with 41.0 uL of mix1'] into plate1[grid:33 site:1] in order [
25	Get DiTis	DiTi 1000ul	
26	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 1)	
27	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 1)	
28	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 1)	
29	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 1)	
30	Dispense	10.00 µl Water free "plate1" (Col. 1, Rows 1-4)	
31	Group End	Spread: 10.0 µL of mix1	
32	Group	Spread: 10.0 µL of mix2	
33	Comment	Spread: 10.0 µL of mix2 (41.6 µL total) from [grid:12 site:1 ['well 2 in mixes : with 41.0 uL of mix2'] into plate1[grid:33 site:1] in order [
34	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 2)	
35	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 2)	
36	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 2)	
37	Aspirate	10.00 µl Water free "mixes" (Col. 1, Row 2)	

38	Dispense		10.00 µl Water free "plate1" (Col. 1, Rows 5-8)
39	Group End	Spread: 10.0 µL of mix2	
40	Group	Spread: 90.0 µL of Buffer	
41	Comment	Spread: 90.0 µL of Buffer (1497.6 µL total) from [grid:53 site:1 ['well 1 in BufferCub : with 1497.0 uL of Buffer '] into plate1[grid:33 si
42	Aspirate		90.00 µl Water free "BufferCub" (Col. 1, Rows 3-6)
43	Dispense		90.00 µl Water free "plate1" (Col. 1, Rows 1-4)
44	Group End	Spread: 90.0 µL of Buffer	
45	Group	Spread: 90.0 µL of Buffer	
46	Comment	Spread: 90.0 µL of Buffer (1497.6 µL total) from [grid:53 site:1 ['well 1 in BufferCub : with 1137.0 uL of Buffer '] into plate1[grid:33 si
47	Aspirate		90.00 µl Water free "BufferCub" (Col. 1, Rows 3-6)
48	Dispense		90.00 µl Water free "plate1" (Col. 1, Rows 5-8)
49	Group End	Spread: 90.0 µL of Buffer	
50	Group	Transfer: 10.0 µL from plate1	

51	Comment	Transfer: 10.0 µL from plate1 [grid:33 site:1] in order [1, 2, 3, 4] into plate2[grid:33 site:3] in order [1, 2, 3, 4]:
52	Aspirate		10.00 µl Water free "plate1" (Col. 1, Rows 1-4)
53	Dispense		10.00 µl Water free "plate-2-moved" (Col. 1, Rows 1-4)
54	Group End	Transfer: 10.0 µL from plate1	
55	Group	Transfer: 10.0 µL from plate1	
56	Comment	Transfer: 10.0 µL from plate1 [grid:33 site:1] in order [5, 6, 7, 8] into plate2[grid:33 site:3] in order [5, 6, 7, 8]:
57	Aspirate		10.00 µl Water free "plate1" (Col. 1, Rows 5-8)
58	Dispense		10.00 µl Water free "plate-2-moved" (Col. 1, Rows 5-8)
59	Group End	Transfer: 10.0 µL from plate1	
60	Group	Spread: 90.0 µL of Buffer	
61	Comment	Spread: 90.0 µL of Buffer (1497.6 µL total) from [grid:53 site:1 ['well 1 in BufferCub : with 777.0 uL of Buffer '] into plate2[grid:33 site
62	Aspirate		90.00 µl Water free "BufferCub" (Col. 1, Rows 3-6)
63	Dispense		90.00 µl Water free "plate-2-moved" (Col. 1, Rows 1-4)

53	Group End		Transfer: 10.0 µL from plate1
60	Group		Spread: 90.0 µL of Buffer
61		Comment	Spread: 90.0 µL of Buffer (1497.6 µL total) from [grid:53 site:1 ['well 1 in BufferCub : with 777.0 uL of Buffer '] into plate2[grid:33 site
62		Aspirate	90.00 µl Water free "BufferCub" (Col. 1, Rows 3-6)
63		Dispense	90.00 µl Water free "plate-2-moved" (Col. 1, Rows 1-4)
64	Group End		Spread: 90.0 µL of Buffer
65	Group		Spread: 90.0 µL of Buffer
66		Comment	Spread: 90.0 µL of Buffer (1497.6 µL total) from [grid:53 site:1 ['well 1 in BufferCub : with 417.0 uL of Buffer '] into plate2[grid:33 site
67		Aspirate	90.00 µl Water free "BufferCub" (Col. 1, Rows 3-6)
68		Dispense	90.00 µl Water free "plate-2-moved" (Col. 1, Rows 5-8)
69	Group End		Spread: 90.0 µL of Buffer
70	Drop DiTis		"DiTi Waste"
71	Group End		Fill plate with mixes

API

The final goal of RobotEvo is to help you to translate your normal laboratory protocol into an script executable by a robot. Thus, the *class Protocol* offers the base to all the custom protocol classes you will write. You will probably derived from *Protocol* one more intermediary protocol class that will define a few characteristic specific to the concrete robot you are using, probably setting some "sensible defaults" in the constructor of that "robot-specific protocol class".

Now, in a concrete protocol derived from that intermediary class, you will define concrete arguments in the constructor, and re-implement the *run()* function. Particularly, you may pass to the constructor the path to the description of the worktable you want to use (this may be an evoware worktable template file or just a working script file).

Normally, by running *run()* the worktable file is parsed and numerous *Labware* objects are created in association with that ´Worktable´. In this function you will then create the *Reagent's you want to manipulate, and also may 'get_labware(label) - where *label* is the unique name given to some object (tip rack, microplate, etc.) in the worktable defined in evoware. Reagent's and 'Labware* objects give you access to individual *Wells*, which are the basic containers of liquid (or *Reagent's) and which, in some circumstances, can be manipulated too. (todo: edit laware's names directly from RobotEvo). After some of those objects are created you can begin to perform the actions that the API 'protocol steps* offers.

How exactly the robot arms pipette (speed, liquid level detection, wet-free, etc.) is defined by liquid classes. The liquid classes are managed internally in an evoware data bank associate with each concrete/physical robot and exposed to the script only by name. One name in one robot may mean something different in another robot or may not be defined at all. Consequently, in RobotEvo the Liquid classes are set by name. The name have to be exactly copied from evoware. Use or create a named liquid class in evoware with all the characteristic you need. Them transfer the name to RobotEvo. Create and transfer to RobotEvo as many Liquid classes as you need.

Principal API: Protocol steps

All these functions are member of the base class `Protocol`, from which all user protocols are derived.

7.1 High level functions:

These are the functions you will use in "every day" protocol programming. They allow you to specify the kind of tips to use and them command the operations you need on your reagents, samples, reactions, etc., almost directly as it states in the steps of your "hand written" original laboratory protocol.

- *tips()*: *how to use tips during the involved instructions.*
- *distribute()*: *some volume of reagent into the wells of the target labware*
- *transfer()*: *from some wells into equal number of target wells*
- *mix()*: *mix by pipetting the content of wells*
- *mix_reagent()*: *mix every aliquot by pipetting*
- *make_pre_mix()*: *put together the components of a PreMixReagent by pipetting*
- *get_tips()*: *mount new or used tips*
- *drop_tips()*: *drop or put back tips*
- *set_first_tip()*: *position of the given tip type to be used next*
- *check_reagents_levels()*: *generate instruction to check all defined reagents*
- *check_reagent_level()*: *generate instruction to check reagent volume*
- *show_check_list()*: *to the operator*
- *comment()*: *add a comment to the script*
- *user_prompt()*: *show a text box to the operator*

7.2 Advanced functions.

Are you doing some advanced protocol development that cannot be efficiently or clearly expressed using the previous High level functions? Then, you may use the following functions.

7.2.1 Atomic functions

These are functions aimed to isolate what a physical robot would make at once: pick some tips, aspirate some liquid, etc. They are simple to understand, but are harder to use in "every day" protocol programming. They may be a great tool to set up your robot and to get an initial familiarization with all the system. Keep in mind that it is now your responsibility to know what robot/protocol "state" are ignored by these new functions. For example, before *aspirate* you will need to mount "by yourself" the tips in the correct position of the used arm, because *aspirate* ignores the higher level with *tips()*. But don't worry, **RobotEvo** still keeps track of the "internal" robot state and will throw errors informing you about most logical mistakes (like in the previous example forgetting to mount the tips). In some cases these functions may be used to construct new high lever functions.

- *pick_up_tip()*: *pick tips of the given type*
- *drop_tip()*
- *aspirate()*: *some volumen from given wells*
- *dispense()*: *some volumen to given wells*

7.2.2 Protocol-structure or state functions

Related to initialization:

- *def_versions()*
- set_paths()
- init_EvoMode()
- *set_defaults()*
- liquid_classes()
- carrier_types()
- allow_labware()
- labware_types()

Related to execution order:

- *use_version()*
- *initialize()*
- *run()*
- pre_check()
- check_list()
- post_check()
- done()

Related to state:

- get_liquid_class()

- get_carrier_type()

- get_labware_type()

- set_EvoMode()

- *set_drop_tips()*

- set_allow_air()

- *reuse_tips()*

- reuse_tips_and_drop()

- preserve_tips()

- preserveing_tips()

- use_preserved_tips()

7.2.3 Other intermediate level functions:

- *aspirate_one()*

- *dispense_one()*

- _dispensemultiwells()

- _aspirate_multi_tips()

- _multidispense_in_replicas()

- comments()

---

class protocol_steps.**Executable**(*GUI=None*, *run_name=None*)

Bases: object

Each executable will need to implement these methods.

__init__(*GUI=None*, *run_name=None*)
Initialize self. See help(type(self)) for accurate signature.

def_versions()
Override this function to define a 'dictionary' of the versions for your Executable, with a name as key and a method as value, which will initialize the Executable to effectively execute that version. You don't need to call this function. It will be used internally during initialization of your derived class.

initialize()
It is called "just in case" to ensure we don't go uninitialized in lazy initializing scenarios.

run()
Here we have accesses to the "internal robot" self.iRobot, with in turn have access to the used Work Table, self.iRobot.worktable from where we can obtain labwares with get_labware(). Overwrite this function and dont call this basic function. This basic function is provided only as an example of "boiled-plate" code

set_defaults()
Set initial values that will not be rest during secondary initializations. The "primary initialization" maybe a light one, like defining the list of versions available. Here, for example, initialize the list of reagents. # todo: make private

use_version(*version: str*)
Select the version to be execute

Parameters **version** (*str*) – the name of the desired version

class protocol_steps.**Pipeline** (*GUI=None*, *protocols=None*, *run_name=None*)

Bases: *protocol_steps.Executable*

Each custom Pipeline need to implement these functions.

__init__ (*GUI=None*, *protocols=None*, *run_name=None*)
Initialize self. See help(type(self)) for accurate signature.

class protocol_steps.**Protocol** (*n_tips=4*, *num_of_samples=96*, *GUI=None*, *work-table_template_filename=None*, *output_filename=None*, *first_tip=None*, *run_name=None*, *tips_type=None*)

Bases: *protocol_steps.Executable*

Base class from which each custom protocol need to be derived, directly or from one of already derived. For example from the already adapted to some generic type robot like Evo200 or from an even more especially adapted like Evo100_FLI. Each newly derived protocol have to optionally override some of the following functions, especially .run().

- High level API:

- App-Structure API:

- Context-options modifiers:

- Lower lever API & "private" functions:

- Atomic API:

__init__ (*n_tips=4*, *num_of_samples=96*, *GUI=None*, *worktable_template_filename=None*, *output_filename=None*, *first_tip=None*, *run_name=None*, *tips_type=None*)

Parameters

- **n_tips** –

- **num_of_samples** –

- **GUI** –

- **worktable_template_filename** –

- **output_filename** –

- **first_tip** –

- **run_name** –

- **tips_type** –

aspirate (*arm: EvoScriPy.robot.Arm = None*, *TIP_MASK: int = None*, *volume: (<class 'float'>, <class 'list'>) = None*, *from_wells: [<class 'EvoScriPy.labware.Well'>] = None*, *liq_class: str = None*)

Atomic operation. Use arm (pipette) with masked (selected) tips to aspirate volume from wells. :param arm: Uses the default Arm (pipette) if None :param TIP_MASK: Binary flag bit-coded (tip1=1, tip8=128) selects tips to use in a multichannel pipette arm.

If None all tips are used. (see Robot.mask_tip[index] and Robot.mask_tips[index])

Parameters

- **volume** – One (the same) for each tip or a list specifying the volume for each tip.

- **from_wells** – list of wells to aspirate from.

- **liq_class** – the name of the Liquid class, as it appears in your own EVOware database. instructions.def_liquidClass if None

aspirate_one(*tip*, *reagent*, *vol=None*, *offset=None*)
  Aspirate vol with ONE tip from reagent

  Parameters

  - **self** –

  - **tip** –

  - **reagent** –

  - **vol** –

  - **offset** –

check_list()
  Tipically

check_reagent_level(*reagent*, *liq_class=None*)
  Select all possible replica of the given reagent and detect the liquid level, contrasting it with the current (expected) volumen in EACH well. Use the given liquid class or the reagent default.

  Parameters

  - **reagent** –

  - **liq_class** –

check_reagents_levels()
  Will emit a liquid level detection on every well occupied by all the reagents defined so fort. Will be executed at the end of self.check_list() but only if self.check_initial_liquid_level is True

comment(*text: str*)
  Add a text comment in the generated script :param text:

consolidate()
  Volumes going to the same destination well are combined within the same tip, so that multiple aspirates can be combined to a single dispense. If there are multiple destination wells, the pipette will never combine their volumes into the same tip. :return:

dispense(*arm: EvoScriPy.robot.Arm = None*, *TIP_MASK: int = None*, *volume: (<class 'float'>, <class 'list'>) = None*, *to_wells: [<class 'EvoScriPy.labware.Well'>] = None*, *liq_class: str = None*)
  Atomic operation. Use arm (pipette) with masked (selected) tips to dispense volume to wells.

  Parameters

  - **arm** – Uses the default Arm (pipette) if None

  - **TIP_MASK** – Binary flag bit-coded (tip1=1, tip8=128) selects tips to use in a multichannel pipette arm. If None all tips are used. (see Robot.mask_tip[index] and Robot.mask_tips[index])

  - **volume** – One (the same) for each tip or a list specifying the volume for each tip.

  - **to_wells** – list of wells to aspirate from.

  - **liq_class** – the name of the Liquid class, as it appears in your own EVOware database. instructions.def_liquidClass if None

dispense_one(*tip*, *reagent*, *vol=None*)
  Dispense vol with ONE tip to reagent

Parameters

- **tip** –

- **reagent** –

- **vol** –

distribute(*volume: float = None*, *reagent: EvoScriPy.reagent.Reagent = None*, *to_labware_region: EvoScriPy.labware.Labware = None*, *optimize: bool = True*, *num_samples: int = None*, *using_liquid_class: (<class 'str'>, <class 'tuple'>) = None*, *TIP_MASK: int = None*, *num_tips: int = None*)

Parameters

- **volume** – if not, volume is set from the default of the source reagent

- **reagent** – Reagent to distribute

- **to_labware_region** – Labware in which the destine well are selected

- **optimize** – minimize zigzag of multi pipetting

- **num_samples** – Priorized !!!! If true reset the selection

- **using_liquid_class** –

- **TIP_MASK** –

- **num_tips** – the number of tips to be used in each cycle of pipetting = all

To distribute a reagent into some wells. This is a high level function with works with the concept of "reagent". This a a concept introduced by RobotEvo that don't exist in EVOware and other similar software. It encapsulated the name, wells occupied by each of the aliquots of the reagent, the volume corresponding to one sample (if any) and the current volume in each aliquot. This function can use multiple of those aliquots to distribute the reagent to the target wells using multiple tips (the maximum will be used if *num_tips* is not set).

By default a number of wells equal to the number of samples set in the protocol will be auto selected in the target labware *to_labware_region*, but this selection can be set explicitly (setting *well.selFlag=True*, for example by calling *to_labware_region.selectOnly(self, sel_idx_list)*). If *num_samples* is set it will rewrite (reset) the selected wells in the target *to_labware_region*.

Please, carefully indicate whether to use "parallel optimization" in the pipetting order by setting *optimize*. (very important unless you are using a full column pipetting arm). Check that the created script don't have conflicts in the order of the samples and the "geometry" of the labware areas (selection of wells) during each pipetting step. This is ease to "see" in the EVOware visual script editor. The generated .protocol.txt can also be used to check this. RobotEvo will detect some errors, but currently not all, assuming the areas are relatively "regular".

The same volume will be transfer to each well. It will be dispensed in only one pass: muss fit into the current tips If the liquid classes (LC) to be used are not explicitly passed the default LC of the reagent will be used. The generated .esc and .gwl can also be used to check this.

A human readable comment will be automatically added to the script with the details of this operation.

drop_tip(*TIP_MASK: int = None*, *DITI_waste: EvoScriPy.labware.Labware = None*, *arm: EvoScriPy.robot.Arm = None*, *airgap_volume: float = 0*, *airgap_speed: int = None*)

Parameters

- **TIP_MASK** – Binary flag bit-coded (tip1=1, tip8=128) selects tips to use in a multichannel pipette arm. If None all tips are used. (see Robot.mask_tip[index] and Robot.mask_tips[index])

- **DITI_waste** – Specify the worktable position for the DITI waste you want to use. You must first put a DITI waste in the Worktable at the required position.

- **arm** – Uses the default Arm (pipette) if None

- **airgap_speed** – int 1-1000. Speed for the airgap in μl/s

- **airgap_volume** – 0 - 100. Airgap in μl which is aspirated after dropping the DITIs

drop_tips (*TIP_MASK=None*)

It will decide to really drop the tips or to put it back in some DiTi rack

Parameters TIP_MASK –

get_tips (*TIP_MASK=None*, *tip_type=None*, *selected_samples=None*)

It will decide to get new tips or to pick back the preserved tips for the selected samples

Parameters

- **TIP_MASK** –

- **tip_type** –

- **selected_samples** –

Returns the TIP_MASK used

initialize ()

It is called "just in case" to ensure we don't go uninitialized in lazy initializing scenarios.

make_pre_mix (*pre_mix: EvoScriPy.reagent.PreMixReagent*, *num_samples: int = None*, *force_replies:*
bool = False)

A preMix is just that: a premix of reagents (aka - components) which have been already defined to add some vol per sample. Uses one new tip per component. It calculates and checks self the minimum and maximum number of replica of the resulting preMix

Parameters

- **pre_mix** ([PreMixReagent](#)) – what to make, a predefined preMix

- **num_samples** (*int*) –

- **force_replies** (*bool*) – use all the preMix predefined replicas

mix (*in_labware_region: EvoScriPy.labware.Labware*, *using_liquid_class: str = None*, *volume: float =*
None, *optimize: bool = True*)

MixReagent the reagents in each of the wells selected *in_labware_region*, *using_liquid_class* and *volume*

Parameters

- **in_labware_region** –

- **using_liquid_class** –

- **volume** –

- **optimize** –

mix_reagent (*reagent: EvoScriPy.reagent.Reagent*, *liq_class: str = None*, *cycles: int = 3*, *maxTips:*
int = 1, *v_perc: int = 90*)

Select all possible replica of the given reagent and mix using the given % of the current vol in EACH well or the max vol for the tip. Use the given "liquid class" or the reagent default.

Parameters

- **reagent** –

- **liq_class** –

- **cycles** –

- **maxTips** –

- **v_perc** – % of the current vol in EACH well to mix

pick_up_tip(*TIP_MASK: int = None, tip_type: (<class 'str'>, <class 'Evo-ScriPy.labware.DITIrackType'>, <class 'EvoScriPy.labware.DITIrack'>, <class 'EvoScriPy.labware.DITIrackTypeSeries'>) = None, arm: EvoScriPy.robot.Arm = None, airgap_volume: float = 0, airgap_speed: int = None*)

Atomic operation. Get new tips. It take a labware type or name instead of the labware itself (DiTi rack) because the real robot take track of the next position to pick, including the rack and the site (the labware). It only need a labware type (tip type) and it know where to pick the next tip. Example:

self.pick_up_tip('DiTi 200 ul') # will pick a 200 ul tip with every tip arm.

Parameters

- **TIP_MASK** – Binary flag bit-coded (tip1=1, tip8=128) selects tips to use in a multichannel pipette arm. If None all tips are used. (see Robot.mask_tip[index] and Robot.mask_tips[index])

- **tip_type** – if None the worktable default DiTi will be used.

- **arm** – Uses the default Arm (pipette) if None

- **airgap_speed** – int 1-1000. Speed for the airgap in μl/s

- **airgap_volume** – 0 - 100. Airgap in μl which is aspirated after dropping the DITIs

reuse_tips(*reuse=True*) → bool

Reuse the tips or drop it and take new after each action?

Parameters reuse (*bool*) –

Returns

run()

Here we have accesses to the "internal robot" self.iRobot, with in turn have access to the used Work Table, self.iRobot.worktable from where we can obtain labwares with get_labware()

set_defaults()

Set initial values that will not be rest during secondary initializations. The "primary initialization" maybe a light one, like defining the list of versions available. Here, for example, initialize the list of reagents. # todo: make private

set_drop_tips(*drop=True*) → bool

Drops the tips at THE END of the whole action? like after distribution of a reagent into various targets :param drop: :return:

set_first_tip(*first_tip: (<class 'int'>, <class 'str'>) = None, tip_type: (<class 'str'>, <class 'Evo-ScriPy.labware.DITIrackType'>) = None*)

Optionally set the Protocol.first_tip, a position in rack, like 42 or 'B06' (optionally including the rack self referenced with a number, like '2-B06', were 2 will be the second rack in the worktable series of the default tip type). Currently, for a more precise set, use directly:

instructions.set_DITI_Counter2(labware=rack, posInRack=first_tip).exec()

show_check_list()

Will show a user prompt with a check list to set all defined reagents: Name, position in the worktable, wells and initial volume (on every well occupied by all the reagents defined so fort. Will be executed at the end of self.check_list() but only if self.show_runtime_check_list is True

tips (*tips_mask=None,*   *tip_type=None,*   *reuse=None,*   *drop=None,*   *preserve=None,*
     use_preserved=None, drop_first=False, drop_last=False, allow_air=None, selected_samples:
     EvoScriPy.labware.Labware = None)
     A contextmanager function which will manage how tips will be used during execution of the dependant
     instructions

> **Parameters**
>
> - **tips_mask** –
>
> - **tip_type** – the type of the tips to be used
>
> - **reuse** (*bool*) – Reuse the tips already mounted? or drop and take new BEFORE each
>   individual action
>
> - **drop** (*bool*) – Drops the tips AFTER each individual action? like after one aspiration
>   and distribute of the reagent into various targets
>
> - **preserve** (*bool*) – puts the tip back into a free place in some rack of the same type
>
> - **use_preserved** – pick the tips back from the previously preserved
>
> - **selected_samples** –
>
> - **allow_air** –
>
> - **drop_first** (*bool*) – Reuse the tips or drop it and take new once BEFORE the whole
>   action
>
> - **drop_last** (*bool*) – Drops the tips at THE END of the whole action

transfer (*from_labware_region:*   *EvoScriPy.labware.Labware,*   *to_labware_region:*   *Evo-*
     ScriPy.labware.Labware, volume: (<class 'int'>, <class 'float'>), using_liquid_class:
     (<class 'str'>, <class 'tuple'>) = None, optimize_from: bool = True, optimize_to: bool
     = True, num_samples: int = None) -> (<class 'EvoScriPy.labware.Labware'>, <class
     'EvoScriPy.labware.Labware'>)

> **Parameters**
>
> - **from_labware_region** (*Labware*) – Labware in which the source wells are located
>   and possibly selected
>
> - **to_labware_region** (*Labware*) – Labware in which the target wells are located and
>   possibly selected
>
> - **volume** (*float*) – if not, volume is set from the default of the source reagent
>
> - **using_liquid_class** – LC or tuple (LC to aspirate, LC to dispense)
>
> - **optimize_from** (*bool*) – use from_labware_region.parallelOrder() to aspirate?
>
> - **optimize_to** (*bool*) – use to_labware_region.parallelOrder() to aspirate?
>
> - **num_samples** (*int*) – Prioritized. If used reset the well selection

> **Returns** a tuple of the labwares used as origin and target with the involved wells selected.

To transfer reagents (typically samples or intermediary reactions) from some wells in the source labware
to the same number of wells in the target labware using the current LiHa arm with maximum number of
tips (of type: *self.worktable.def_DiTi_type*, which can be set 'with self. *tips()* (tip_type = myTipsRack-
Type)'). # todo: count for 'broken' tips

The number of "samples" may be explicitly indicated in which case will be assumed to begin from the
first well of the labware. Alternatively the wells in the source or target or in both may be previously di-
rectly "selected" (setting *well.selFlag=True*, for example by calling *from_labware_region.selectOnly(self,
sel_idx_list)*), in which case transfer the minimum length selected. If no source wells are selected this

---

function will auto select the protocol's *self.num_of_samples* number of wells in the source and target labwares. Please, carefully indicate whether to use "parallel optimization" in the pipetting order for both source and target by setting *optimizeFrom* and *optimizeTo*. (very important unless you are using a full column pipetting arm). Check that the created script don't have conflicts in the order of the samples and the "geometry" of the labware areas (selection of wells) during each pipetting step. This is ease to "see" in the EVOware visual script editor. The generated .protocol.txt can also be used to check this. RobotEvo will detect some errors, but currently not all, assuming the areas are relatively "regular".

The same volume will be transfer from each well. It will be aspirated/dispensed in only one pass: muss fit into the current tips todo ?! If no *volume* is indicated then the volume expected to be in the first selected well will be used.

If the liquid classes (LC) to be used are not explicitly passed the default LC in the first well of the current pipetting step will be used. The generated .esc and .gwl can also be used to check this.

A human readable comment will be automatically added to the script with the details of this operation.

Warning: modify the selection of wells in both source and target labware to reflect the wells actually used

user_prompt (*text: str*, *sound: bool = True*, *close_time: int = -1*)
Interrupt pippeting to popup a message box to the operator.

> **Parameters**
>
> - **text** – the text in the box
>
> - **sound** – Should add an acustic signal?
>
> - **close_time** – time to wait for automatic closing the box: the operator can "manually"

close this box at any time, and this will be the only way to close it if the default -1 is used, which
cause no automatic closing.

Reagent - a fundamental concept

A *Reagent* is a fundamental concept in RobotEvo programming. It makes possible to define a protocol in a natural way, matching what a normal laboratory's protocol indicates. Defines a named homogeneous liquid solution, the wells it occupy, the initial amount needed to run the protocol (auto calculated), and how much is needed per sample, if applicable. It is also used to define samples, intermediate reactions and products. It makes possible a robust tracking of all actions and a logical error detection, while significantly simplifying the programming of non trivial protocols.

todo: implement units for volume, concentration, etc.

8.1 Main classes and functions:

8.1.1 Abstract information classes:

- `MixComponent`: like an item in some table summarizing components of some Mix.

- `PreMixComponent`: like an item in some table summarizing components of some PreMix.

- `Primer`: like an item in some table summarizing primer sequences, synthesis, etc.

- `PrimerMixComponent`: like an item in a table describing Primer Mixes for some PCRs.

- `PrimerMix`: like a table describing Primer Mixes for some PCRs

- `PCRMasterMix`: like an item in some table summarizing PCR Master Mixes for some PCR experiment

- `PCReaction`: like an item in some table summarizing reactions in a PCR experiment

- `PCRexperiment`: like an item in some table summarizing PCR experiments

8.1.2 Robot classes:

- `Reagent`: homogeneous liquid solution in some wells

- `MixReagent`: a Reagent composed of other Reagents

- *Dilution*

- *PreMixReagent*: A pre-MixReagent of otherwise independent reagents

- *PrimerReagent*: Manipulate a Primer Reagent on a robot.

- *PrimerMixReagent*: Manipulate a Primer-Mix Reagent on a robot.

- *PCRMasterMixReagent*: Manipulate a PCR Master-Mix Reagent on a robot.

- *PCReactionReagent*

- *PCRexperimentRtic*: Organize a PCR setup on a robot.

class reagent.**Dilution**(*name: str, diluent: reagent.Reagent, labware: (<class 'Evo-ScriPy.labware.Labware'>, <class 'str'>, []) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'EvoScriPy.labware.Well'>]) = None, components: [<class 'reagent.DilutionComponentReagent'>] = None, num_of_aliquots: int = None, minimize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>, <class 'str'>)) = None, excess: float = None, initial_vol: float = 0.0, min_vol: float = 0.0, fill_limit_aliq: float = 100, concentration: float = None*)*

   Bases: *reagent.MixReagent*

   A Reagent composed of others, diluted Reagents, that the robot may prepare.

   __init__(*name: str, diluent: reagent.Reagent, labware: (<class 'EvoScriPy.labware.Labware'>, <class 'str'>, []) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'EvoScriPy.labware.Well'>]) = None, components: [<class 'reagent.DilutionComponentReagent'>] = None, num_of_aliquots: int = None, minimize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>, <class 'str'>)) = None, excess: float = None, initial_vol: float = 0.0, min_vol: float = 0.0, fill_limit_aliq: float = 100, concentration: float = None*)*

   Parameters

   - **name** –

   - **labware** –

   - **wells** –

   - **components** –

   - **num_of_aliquots** –

   - **minimize_aliquots** –

   - **def_liq_class** –

   - **excess** –

   - **initial_vol** –

   - **min_vol** –

   - **fill_limit_aliq** –

   - **concentration** –

   components = None
       list of reagent components

class reagent.**DilutionComponentReagent**(*reagent: reagent.Reagent, dilution: float = None, final_conc: float = None*)*

   Bases: *reagent.MixComponentReagent*

Components of some Dilution.

__init__ (*reagent: reagent.Reagent, dilution: float = None, final_conc: float = None*)

> **Parameters**
>
> > • **reagent** –
> >
> > • **dilution** –
> >
> > • **final_conc** –

class reagent.**MixComponent** (*name: str, id_: str = None, init_conc: float = None, final_conc: float = None, volume: float = None*)

Bases: object

Represent abstract information, like an item in some table summarizing components of some MixReagent. todo: introduce diluent? - final_conc == None ? final_conc == init_conc ?

__init__ (*name: str, id_: str = None, init_conc: float = None, final_conc: float = None, volume: float = None*)

> **Parameters**
>
> > • **id** –
> >
> > • **name** –
> >
> > • **init_conc** – todo Really??
> >
> > • **final_conc** –

class reagent.**MixComponentReagent** (*reagent: reagent.Reagent, volume: float = None*)

Bases: object

Components of some MixReagent. This is not just a Reagent, but some "reserved" volume of some Reagent.

__init__ (*reagent: reagent.Reagent, volume: float = None*)

> **Parameters**
>
> > • **reagent** –
> >
> > • **volume** –

class reagent.**MixReagent** (*name: str, labware: (<class 'EvoScriPy.labware.Labware'>, <class 'str'>, []) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'EvoScriPy.labware.Well'>]) = None, components: [<class 'reagent.MixComponentReagent'>] = None, num_of_aliquots: int = None, minimize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>, <class 'str'>)) = None, excess: float = 1.0, initial_vol: float = 0.0, min_vol: float = 0.0, fill_limit_aliq: float = 100, concentration: float = None*)

Bases: *reagent.Reagent*

A Reagent composed of other Reagents, that the robot may prepare.

__init__ (*name: str, labware: (<class 'EvoScriPy.labware.Labware'>, <class 'str'>, []) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'EvoScriPy.labware.Well'>]) = None, components: [<class 'reagent.MixComponentReagent'>] = None, num_of_aliquots: int = None, minimize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>, <class 'str'>)) = None, excess: float = 1.0, initial_vol: float = 0.0, min_vol: float = 0.0, fill_limit_aliq: float = 100, concentration: float = None*)

> **Parameters**
>
> > • **name** –

- **labware** –

- **wells** –

- **components** –

- **num_of_aliquots** –

- **minimize_aliquots** –

- **def_liq_class** –

- **excess** –

- **initial_vol** –

- **min_vol** –

- **fill_limit_aliq** –

- **concentration** –

components = None
list of reagent components

exception reagent.**NoReagentFound**(*reagent_name: str*, *error: str*)
Bases: Exception

__init__(*reagent_name: str*, *error: str*)
Initialize self. See help(type(self)) for accurate signature.

class reagent.**PCRMasterMix**(*name*, *id_=None*, *reaction_vol=25*, *sample_vol=5*, *components=None*,
diluent=None, *title=None*)
Bases: object

Represent abstract information, like an item in some table summarizing PCR Master Mixes for some PCR
experiment

__init__(*name*, *id_=None*, *reaction_vol=25*, *sample_vol=5*, *components=None*, *diluent=None*, *title=None*)

Parameters

- **name** –

- **id** –

- **reaction_vol** –

- **sample_vol** –

- **components** –

- **title** –

ids = {}
connect each existing PCR master mix ID with the corresponding PCRMasterMix

names = {}
connect each existing PCR master mix name with the corresponding PCRMasterMix

class reagent.**PCRMasterMixReagent**(*pcr_mix: reagent.PCRMasterMix, mmix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), num_of_samples: int, pos=None, num_of_aliquots=None, initial_vol=None, def_liq_class=None, excess=None, fill_limit_aliq=None, kit_rack: (<class 'Evo-ScriPy.labware.Labware'>, <class 'list'>) = None, primer_mix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, primer_rack: (<class 'Evo-ScriPy.labware.Labware'>, <class 'list'>) = None*)

Bases: *reagent.PreMixReagent*

Manipulate a PCR Master-Mix Reagent on a robot.

__init__(*pcr_mix: reagent.PCRMasterMix, mmix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), num_of_samples: int, pos=None, num_of_aliquots=None, initial_vol=None, def_liq_class=None, excess=None, fill_limit_aliq=None, kit_rack: (<class 'Evo-ScriPy.labware.Labware'>, <class 'list'>) = None, primer_mix_rack: (<class 'Evo-ScriPy.labware.Labware'>, <class 'list'>) = None, primer_rack: (<class 'Evo-ScriPy.labware.Labware'>, <class 'list'>) = None*)

Construct a robot-usable PCRMasterMixReagent from an abstract PCRMasterMix. It is always constructed - no reuse of old aliquots: contains instable components.

> **Parameters**
>
> > - **primer_mix_rack** –
> >
> > - **primer_rack** –
> >
> > - **pos** –
> >
> > - **num_of_aliquots** –
> >
> > - **initial_vol** –
> >
> > - **def_liq_class** –
> >
> > - **fill_limit_aliq** –
> >
> > - **pcr_mix** –
> >
> > - **mmix_rack** –
> >
> > - **kit_rack** –
> >
> > - **excess** –

class reagent.**PCReaction**(*rol, sample=None, targets=None, mix: reagent.PCRMasterMix = None, replica=None, row=None, col=None, vol=None*)

Bases: object

Represent abstract information, like an item in some table summarizing reactions in a PCR experiment

__init__(*rol, sample=None, targets=None, mix: reagent.PCRMasterMix = None, replica=None, row=None, col=None, vol=None*)

Initialize self. See help(type(self)) for accurate signature.

class reagent.**PCReactionReagent**(*pcr_reaction: reagent.PCReaction, plate: Evo-ScriPy.labware.Labware*)

Bases: *reagent.Reaction*

__init__(*pcr_reaction: reagent.PCReaction, plate: EvoScriPy.labware.Labware*)

> **Parameters**
>
> > - **pcr_reaction** –

> - **plates** –

class reagent.**PCRexperiment**(*id_=None*, *name=None*, *ncol=0*, *nrow=0*)

> Bases: object
>
> Represent abstract information, like an item in some table summarizing PCR experiments
>
> **__init__**(*id_=None*, *name=None*, *ncol=0*, *nrow=0*)
>> A linear rack have just one roe and many columns
>>
>>> **Parameters**
>>>
>>>> - **id** –
>>>>
>>>> - **name** –
>>>>
>>>> - **ncol** –
>>>>
>>>> - **nrow** –
>
> **mixes = {}**
>> connect each PCR master mix with a list of well reactions
>
> **pcr_reactions = None**
>> list of PCRReaction to create organized in rows with columns
>
> **samples = None**
>> connect each sample with a list of well reactions
>
> **targets = None**
>> connect each target with a list of PCR reactions well

class reagent.**PCRexperimentRtic**(*pcr_exp: (<class 'reagent.PCRexperiment'>, <class 'list'>), plates: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), kit_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), mmix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, primer_mix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, primer_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, protocol=None*)

> Bases: object
>
> Organize a PCR setup on a robot. From a list of abstract information about PCR plate/experiments creates sufficient volume of each of the *PCRMasterMixReagent* listed in the global PCRexperiment.mixes
>
> **__init__**(*pcr_exp: (<class 'reagent.PCRexperiment'>, <class 'list'>), plates: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), kit_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>), mmix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, primer_mix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, primer_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None, protocol=None*)
>
>> **Parameters**
>>
>>> - **pcr_exp** – [*PCRexperiment*] abstarct information about the "plate" PCR experiemnts
>>>
>>> - **plates** – [*labware.Labware*] where to set the PCR reactions.
>>>
>>> - **kit_rack** – [racks] in the prefered order to put the PCR kit reagents (stocks solutions)
>>>
>>> - **mmix_rack** – [racks] in the prefered order to put the PCR mastermix reagents specially created for these experiments

- **primer_mix_rack** – [racks] in the prefered order to put the primer mix reagents (stocks solutions)s

- **primer_rack** – [racks] in the prefered order to put the primers reagents (stocks solutions)s

- **protocol** – who invoke this PCR, provide a worktable and the rest of the "environment"

mixes = None
> connect each *PCRMasterMix* in the experiment with the PCR wells into which will be pippeted

pcr_exp = None
> abstract info

class reagent.**PreMixComponent**(*name: str*, *volpersample: float*, *id_: str = None*, *init_conc: float = None*, *final_conc: float = None*, *volume: float = None*)
> Bases: *reagent.MixComponent*

Represent abstract information, like an item in some table summarizing components of some PreMixReagent. An special case of MixComponent, for which volume is calculated on the basis of "number of samples" and volume_per_sample

__init__(*name: str*, *volpersample: float*, *id_: str = None*, *init_conc: float = None*, *final_conc: float = None*, *volume: float = None*)

> **Parameters**
>
> - **id** –
>
> - **name** –
>
> - **init_conc** – todo Really??
>
> - **final_conc** –

class reagent.**PreMixReagent**(*name*, *labware: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>)*, *components*, *pos=None*, *num_of_aliquots=None*, *initial_vol=None*, *def_liq_class=None*, *excess=None*, *fill_limit_aliq=None*, *num_of_samples=None*)
> Bases: *reagent.Reagent*

A pre-MixReagent of otherwise independent reagents to be pippeted together for convenience, but that could be pippeted separately. todo: make this a special case of MixReagent, for which everything ? is calculated on the basis of "number of samples"

__init__(*name*, *labware: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>)*, *components*, *pos=None*, *num_of_aliquots=None*, *initial_vol=None*, *def_liq_class=None*, *excess=None*, *fill_limit_aliq=None*, *num_of_samples=None*)

> **Parameters**
>
> - **name** –
>
> - **labware** –
>
> - **components** – list of reagent components
>
> - **pos** –
>
> - **num_of_aliquots** –
>
> - **initial_vol** –
>
> - **def_liq_class** –
>
> - **excess** –

> > - **fill_limit_aliq** –
> >
> > - **num_of_samples** –

components = None
> list of reagent components

init_vol(*num_samples=None*, *initial_vol=None*)
> update my self.volpersample from the already updated self.components.volpersample, possibly with updated num_samples and initial_vol WARNING !! call this only after the update of the components (if need) :param num_samples: :param initial_vol:

class reagent.**Primer**(*name: str*, *seq: str*, *proposed_stock_conc: float = 100*, *id_: str = None*, *prepared: float = None*, *mass: float = None*, *moles: float = None*, *molec_w: float = None*, *mod_5p: str = None*, *mod_3p: str = None*, *id_synt: str = None*, *kws: list = None*, *diluent: str = 'TE 1x'*)
Bases: object

Represent abstract information, like an item in some table summarizing primer sequences, synthesis, etc.

__init__(*name: str*, *seq: str*, *proposed_stock_conc: float = 100*, *id_: str = None*, *prepared: float = None*, *mass: float = None*, *moles: float = None*, *molec_w: float = None*, *mod_5p: str = None*, *mod_3p: str = None*, *id_synt: str = None*, *kws: list = None*, *diluent: str = 'TE 1x'*)

> **Parameters**
>
> > - **name** –
> >
> > - **seq** –
> >
> > - **proposed_stock_conc** –
> >
> > - **id** –
> >
> > - **prepared** –
> >
> > - **mass** –
> >
> > - **moles** –
> >
> > - **molec_w** –
> >
> > - **mod_5p** –
> >
> > - **mod_3p** –
> >
> > - **id_synt** –
> >
> > - **kws** –
> >
> > - **diluent** –

ids = {}
> connect each existing Primer ID with the corresponding Primer

ids_synt = {}
> connect each existing Primer synthesis ID with the corresponding Primer

key_words = {}
> connect each existing Primer key_word with the corresponding list of Primer

names = {}
> connect each existing Primer name with the corresponding list of Primer

seqs = {}
> connect each existing Primer sequence with the corresponding list of Primer

---

class reagent.**PrimerMix**(*name*, *id_=None*, *conc=10.0*, *prepared=None*, *components=None*, *ref_vol=None*, *diluent=None*, *kws=None*, *super_mix=False*)

Bases: object

Represent abstract information, like a table describing Primer Mixes for some PCRs

__init__(*name*, *id_=None*, *conc=10.0*, *prepared=None*, *components=None*, *ref_vol=None*, *diluent=None*, *kws=None*, *super_mix=False*)

Parameters

- **name** –
- **id** –
- **conc** –
- **prepared** –
- **components** –
- **ref_vol** –
- **diluent** –
- **kws** –
- **super_mix** –

ids = {}
connect each existing Primer mix ID with the corresponding PrimerMix

key_words = {}
connect each existing Primer mix key_word with the corresponding list of PrimerMix

names = {}
connect each existing Primer mix name with the corresponding list of PrimerMix

class reagent.**PrimerMixComponent**(*id_=None*, *name=None*, *init_conc=None*, *final_conc=None*, *super_mix: bool = False*)

Bases: *reagent.MixComponent*

Represent abstract information, like an item in a table describing Primer Mixes for some PCRs. It can be a primer, another primer mix or the diluent

__init__(*id_=None*, *name=None*, *init_conc=None*, *final_conc=None*, *super_mix: bool = False*)

Parameters

- **id** –
- **name** –
- **init_conc** –
- **final_conc** –
- **super_mix** –

class reagent.**PrimerMixReagent**(*primer_mix: reagent.PrimerMix*, *primer_mix_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>)*, *pos=None*, *num_of_aliquots=None*, *initial_vol=None*, *def_liq_class=None*, *excess=None*, *fill_limit_aliq=None*, *primer_rack: (<class 'EvoScriPy.labware.Labware'>, <class 'list'>) = None*)

Bases: *reagent.PreMixReagent*

Manipulate a Primer-MixReagent Reagent on a robot.

__init__(*primer_mix: reagent.PrimerMix*, *primer_mix_rack: (<class 'Evo-ScriPy.labware.Labware'>*, *<class 'list'>)*, *pos=None*, *num_of_aliquots=None*, *initial_vol=None*, *def_liq_class=None*, *excess=None*, *fill_limit_aliq=None*, *primer_rack: (<class 'EvoScriPy.labware.Labware'>*, *<class 'list'>) = None*)

Construct a robot-usable PrimerMixReagent from an abstract PrimerMix. You can reuse "old" aliquots by passing primer_mix.prepared volume > 0. If no primer_mix.prepared volume is passed, or if it is not sufficient, a set of primer reagents will be created.

> **Parameters**
>
> - **primer_mix** –
>
> - **primer_mix_rack** –
>
> - **pos** –
>
> - **num_of_aliquots** –
>
> - **initial_vol** –
>
> - **def_liq_class** –
>
> - **excess** –
>
> - **fill_limit_aliq** –
>
> - **primer_rack** –

class reagent.**PrimerReagent**(*primer: reagent.Primer*, *primer_rack: (<class 'Evo-ScriPy.labware.Labware'>*, *<class 'list'>)*, *pos=None*, *initial_vol=None*, *PCR_conc=0.8*, *stk_conc=100*, *def_liq_class=None*, *fill_limit_aliq=None*, *excess=None*)

Bases: *reagent.MixReagent*

Manipulate a Primer Reagent on a robot.

__init__(*primer: reagent.Primer*, *primer_rack: (<class 'EvoScriPy.labware.Labware'>*, *<class 'list'>)*, *pos=None*, *initial_vol=None*, *PCR_conc=0.8*, *stk_conc=100*, *def_liq_class=None*, *fill_limit_aliq=None*, *excess=None*)

Construct a robot-usable PrimerReagent from an abstract Primer. You can reuse "old" aliquots by passing primer.prepared volume > 0. If no primer.prepared volume is passed, it will be prepared.

> **Parameters**
>
> - **primer** –
>
> - **primer_rack** –
>
> - **pos** –
>
> - **initial_vol** –
>
> - **PCR_conc** –
>
> - **stk_conc** –
>
> - **def_liq_class** –
>
> - **fill_limit_aliq** –
>
> - **excess** –

class reagent.**Reaction**(*name*, *labware*, *components: [<class 'reagent.Reagent'>] = None*, *track_sample=None*, *pos=None*, *num_of_aliquots=1*, *def_liq_class=None*, *excess=None*, *initial_vol=0*)

Bases: *reagent.Reagent*

todo: make this a MixReagent, with diluent too ?

__init__(*name, labware, components: [<class 'reagent.Reagent'>] = None, track_sample=None,*
pos=None, num_of_aliquots=1, def_liq_class=None, excess=None, initial_vol=0)
This is a named set of aliquots of an homogeneous solution. Put a reagent into labware wells, possible
distributed into aliquots and set the amount to be used for each sample, if applicable. Each reagent is
added to a list of reagents of the worktable were the labware is. The specified excess in % will be calcu-
lated/expected. A default excess of 4% will be assumed if None is indicated. A minimal needed volume
will be calculated based on either the number of samples and the volume per sample to use or the volume
per single use. This can be forced setting min_vol. A minimal number of replicas (wells, aliquots) will be
calculated based on the minimal volume, taking into account the maximum allowed volume per well and
the excess specified. Aliquots will be filled not more than the percent of the well volumen indicated by
fill_limit_aliq.

> **Parameters**
>
> - **name** – Reagent name. Ex: "Buffer 1", "forward primer", "IC MS2"
>
> - **labware** – *labware.Labware* or his label in the worktable; if None will be deduced
>   from *wells*.
>
> - **volpersample** – how much is needed per sample, if applicable, in uL
>
> - **single_use** – Not a "per sample" multiple use? Set then here the volume for one single
>   use
>
> - **wells** – or offset to begging to put replica. If None will try to assign consecutive wells
>   in labware
>
> - **num_of_aliquots** – def min_num_of_replica(), number of replicas
>
> - **def_liq_class** – the name of the Liquid class, as it appears in your own EVOware
>   database. instructions.def_liquidClass if None

:param excess; in % :param initial_vol: is set for each replica. If default (=None) is calculated als min-
imum. :param min_vol: force a minimum volume to be expected or prepared. :param fill_limit_aliq:
maximo allowed volume in % of the wells capacity :param num_of_samples: if None, the number of sam-
ples of the current protocol will be assumed :param minimize_aliquots: use minimal number of aliquots?
Defaults to *Reagent.use_minimal_number_of_aliquots*,

> This default value can be temporally change by setting that global.

class reagent.**Reagent**(*name: str, labware: (<class 'EvoScriPy.labware.Labware'>, <class 'str'>,*
[]) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'Evo-
ScriPy.labware.Well'>]) = None, num_of_aliquots: int = None, mini-
mize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>,
<class 'str'>)) = None, volpersample: float = 0.0, num_of_samples: int =
None, single_use: float = None, excess: float = None, initial_vol: (<class
'float'>, <class 'list'>) = 0.0, min_vol: float = 0.0, fill_limit_aliq: float =
100, concentration: float = None)
Bases: object

A Reagent is a fundamental concept in RobotEvo programming. It makes possible to define a protocol in a
natural way, matching what a normal laboratory's protocol indicates. Defines a named homogeneous liquid
solution, the wells it occupy, the initial amount needed to run the protocol (auto calculated), and how much
is needed per sample, if applicable. It is also used to define samples, intermediate reactions and products. It
makes possible a robust tracking of all actions and a logical error detection, while significantly simplifying the
programming of non trivial protocols.

__init__ (*name: str, labware: (<class 'EvoScriPy.labware.Labware'>, <class 'str'>, []) = None, wells: (<class 'int'>, [<class 'int'>], [<class 'EvoScriPy.labware.Well'>]) = None, num_of_aliquots: int = None, minimize_aliquots: bool = None, def_liq_class: (<class 'str'>, (<class 'str'>, <class 'str'>)) = None, volpersample: float = 0.0, num_of_samples: int = None, single_use: float = None, excess: float = None, initial_vol: (<class 'float'>, <class 'list'>) = 0.0, min_vol: float = 0.0, fill_limit_aliq: float = 100, concentration: float = None*)

This is a named set of aliquots of an homogeneous solution. Put a reagent into labware wells, possible distributed into aliquots and set the amount to be used for each sample, if applicable. Each reagent is added to a list of reagents of the worktable were the labware is. The specified excess in % will be calculated/expected. A default excess of 4% will be assumed if None is indicated. A minimal needed volume will be calculated based on either the number of samples and the volume per sample to use or the volume per single use. This can be forced setting min_vol. A minimal number of replicas (wells, aliquots) will be calculated based on the minimal volume, taking into account the maximum allowed volume per well and the excess specified. Aliquots will be filled not more than the percent of the well volumen indicated by fill_limit_aliq.

> **Parameters**
>
> - **name** – Reagent name. Ex: "Buffer 1", "forward primer", "IC MS2"
>
> - **labware** – *labware.Labware* or his label in the worktable; if None will be deduced from *wells*.
>
> - **volpersample** – how much is needed per sample, if applicable, in uL
>
> - **single_use** – Not a "per sample" multiple use? Set then here the volume for one single use
>
> - **wells** – or offset to begging to put replica. If None will try to assign consecutive wells in labware
>
> - **num_of_aliquots** – def min_num_of_replica(), number of replicas
>
> - **def_liq_class** – the name of the Liquid class, as it appears in your own EVOware database. instructions.def_liquidClass if None

:param excess; in % :param initial_vol: is set for each replica. If default (=None) is calculated als minimum. :param min_vol: force a minimum volume to be expected or prepared. :param fill_limit_aliq: maximo allowed volume in % of the wells capacity :param num_of_samples: if None, the number of samples of the current protocol will be assumed :param minimize_aliquots: use minimal number of aliquots? Defaults to *Reagent.use_minimal_number_of_aliquots*,

> This default value can be temporally change by setting that global.

init_vol (*num_samples=None, initial_vol=None*)

To initialize the among of reagent in each well. First put what the user inform he had put, then put additionally the minimum the protocol need. :param num_samples: :param initial_vol: :return:

min_num_of_replica (*num_of_samples: int = None*) → int

A minimal number of replicas (wells, aliquots) will be calculated based on the minimal volume, taking into account the maximum allowed volume per well and the excess specified. :param num_of_samples: :return:

min_vol (*num_samples=None, volume: float = None, add_volume: float = None*) → float

A minimal volume will be calculated based on either the number of samples and the volume per sample to use (todo or the volume per single use.)??? :param num_samples: :return:

need_vol = None

calculated volume needed during the execution of the protocol

---

put_min_vol (*num_samples=None*)

    Force you to put an initial volume of reagent that can be used to distribute into samples, aspiring equal number of complete doses for each sample from each replica, exept the firsts replicas that can be used to aspirate one more dose for the last/rest of samples. That is: all replica have equal volumen (number) of doses or the firsts have one more dose :param num_samples: :return:

Worktable and labwares

- `Worktable`
- `Worktable.Location`
- *`Labware.Type`* + Specialized types:
    - `DiTiRackType`
    - *`CuvetteType`*
- *`Labware.Type.Series`*
- *`Labware`* + Specialized labwares:
    - `DitiRack`
    - *`Cuvette`*

class labware.**Carrier**(*carrier_type: labware.Carrier.Type*, *grid: int*, *label: str = None*, *worktable: labware.WorkTable = None*)

    Bases: `object`

    Collection of Labwares sites, filled with labwares. . .

    class Type(*name*, *idx: int = None*, *widht_in_grids: int = None*, *n_sites: int = None*)

        Bases: `object`

        __init__(*name*, *idx: int = None*, *widht_in_grids: int = None*, *n_sites: int = None*)
            Initialize self. See help(type(self)) for accurate signature.

    class Types(*carrier_file: pathlib.Path*)

        Bases: `object`

        __init__(*carrier_file: pathlib.Path*)
            Initialize self. See help(type(self)) for accurate signature.

        add_type(*carrier_type*)

        parse_file(*carrier_file=None*)

  __init__(*carrier_type: labware.Carrier.Type*, *grid: int*, *label: str = None*, *worktable: lab-ware.WorkTable = None*)
    Initialize self. See help(type(self)) for accurate signature.

  add_labware(*labware*, *site*)

class labware.**Cuvette**(*type*, *location*, *label=None*)
  Bases: *labware.Labware*

  __init__(*type*, *location*, *label=None*)

    Parameters

      • **type** –

      • **label** –

      • **location** –

  autoselect(*offset=0*, *maxTips=1*, *replys=1*)

    Parameters

      • **offset** –

      • **maxTips** –

      • **replys** –

    Returns

  init_wells()

class labware.**CuvetteType**(*name*, *nRow*, *max_vol*, *nCol=1*)
  Bases: labware.Type

  __init__(*name*, *nRow*, *max_vol*, *nCol=1*)
    Initialize self. See help(type(self)) for accurate signature.

  create_labware(*loc*, *label*)

class labware.**DITIrack**(*type: labware.DITIrackType*, *location: labware.WorkTable.Location*, *label: str*)
  Bases: *labware.Labware*

  Objects of this class represent physical objects (with location) of some type Labware.DITIrackType

  __init__(*type: labware.DITIrackType*, *location: labware.WorkTable.Location*, *label: str*)

    Parameters

      • **type** –

      • **location** –

      • **label** –

      • **worktable** –

  fill(*beg=1*, *end=None*)

  find_new_tips(*number_tips) -> (<class 'bool'>[, <class 'labware.Tip'>]*)
    Return existing tips. May be only partially. Just to know there are tips

  pick_up(*TIP_MASK*) → [<class 'labware.usedTip'>]
    Low level. Part of the job have been already done: the rack self hat already the source tip-wells selected.
    We need to return these tips.

    Parameters TIP_MASK –

retire_new_tips(*number_tips*) → [<class 'labware.Tip'>]

> Return removed tips. May be only partially. Low Level !!! To be called only by implementations of low level Instruction.actualize_robot_state as part of series.retire_new_tips as response to getDITI2

set_DITI_counter(*posInRack*, *lastPos=False*)

set_back(*TIP_MASK*, *tips*)

> Low level. Part of the job have been already done: tips is a list of the tips in the robot arm, passed here just to prevent a call and a link back to the robot. And the rack self hat already the target tip-wells selected.

> > **Parameters**
> >
> > - **TIP_MASK** –
> >
> > - **labware_selection** –
> >
> > - **tips** –

class labware.**DITIrackType**(*name*, *nRow=8*, *nCol=12*, *max_vol=None*, *portrait=False*)

> Bases: labware.Type

__init__(*name*, *nRow=8*, *nCol=12*, *max_vol=None*, *portrait=False*)

> Initialize self. See help(type(self)) for accurate signature.

create_labware(*loc*, *label*)

create_series(*labware: labware.Labware*)

class labware.**DITIrackTypeSeries**(*labware: labware.Labware*)

> Bases: labware.Series

__init__(*labware: labware.Labware*)

> Initialize self. See help(type(self)) for accurate signature.

find_new_tips(*TIP_MASK) -> (<class 'bool'>*[*, <class 'labware.Tip'>*])

> > **Parameters**
> >
> > - **TIP_MASK** –
> >
> > - **lastPos** –
> >
> > **Returns**

refill_next_rack(*worktable=None*)

retire_new_tips(*TIP_MASK*)

> A response to a get_tips: the tips have to be removed from the rack and only after that can appear mounted in the robot arm to pipette. The tips are removed at the "current" position, the position where begin the fresh tips, with is maintained internally by the robot and is unknown to the user

class labware.**DITIwaste**(*type*, *location*, *label=None*)

> Bases: *labware.Labware*

__init__(*type*, *location*, *label=None*)

> > **Parameters**
> >
> > - **type** –
> >
> > - **label** –
> >
> > - **location** –

waste(*tips*)

class labware.**DITIwasteType**(*name*, *capacity=480*)
Bases: labware.Type

__init__(*name*, *capacity=480*)
Initialize self. See help(type(self)) for accurate signature.

create_labware(*loc*, *label*)

class labware.**Frezeer**
Bases: *labware.WorkTable*

__init__()
Initialize self. See help(type(self)) for accurate signature.

class labware.**Labware**(*type: labware.Labware.Type, label: str, location: lab-
ware.WorkTable.Location = None*)
Bases: object

class Position(*row*, *col=1*)
Bases: object

__init__(*row*, *col=1*)
Initialize self. See help(type(self)) for accurate signature.

to_name()

class Type(*name*, *nRow*, *nCol=1*, *max_vol=None*)
Bases: object

class Series(*labware*)
Bases: object

__init__(*labware*)
Initialize self. See help(type(self)) for accurate signature.

add(*labware*)

remove(*labware*)

static set_current_next_to(*labware*)

set_next()
Set current to the next of self.current :rtype: (Labware, bool) = (the next labware , serie's current
has rotated to the first :param labware:

show_next(*labware=None*)
return next to self.current :rtype: (Labware, bool) = (the next labware , serie's current has rotated
to the first :param labware:

static show_next_to(*labware*)

__init__(*name*, *nRow*, *nCol=1*, *max_vol=None*)
Initialize self. See help(type(self)) for accurate signature.

create_labware(*loc*, *label*)

create_series(*labware*)

size() → int

__init__(*type: labware.Labware.Type, label: str, location: labware.WorkTable.Location = None*)

Parameters

- **type** –

- **label** –

> - **location** –

autoselect (*offset=0*, *maxTips=1*, *replys=1*)

> Parameters
>
> > - **offset** –
> >
> > - **maxTips** –
> >
> > - **replys** –
>
> Returns

clearSelection ()

static create (*labw_t_name: str*, *loc: labware.WorkTable.Location*, *label: str*)

find_free_wells (*n=1*, *init_pos=0*) *-> (<class 'bool'>* $\big[$*, <class 'labware.Well'> *$\big]$*)*

init_wells ()

moveParallel (*pos*, *offset*)

newOffset (*pos*, *offset*)

newPosition (*pos*, *offset*)

offset (*row_pos*, *col=1*)

offsetAtParallelMove (*step*, *n_tips*)

offsetFromName (*wellName*)

parallelOrder (*n_tips*, *original=None*)

posAtParallelMove (*step*, *n_tips*)

position (*offset*)

put (*reagent*, *pos=None*, *num_of_aliquots=None*) → list

> Put a reagent with replicas in the given wells positions of this labware, and return a list of the wells used
>
> > Parameters
> >
> > > - **reagent** –
> > >
> > > - **pos** – [wells]; if int or [int] will be assumed 1-based not 0-based
> > >
> > > - **num_of_aliquots** – number of replicas
> >
> > Returns

select (*sel_idx_list*)

selectAll ()

selectOnly (*sel_idx_list*)

selected () → list

> Returns list of the selected well offset

selected_wells ()

types = {'24 Pos Eppi Tube Rack': 6, '96 Well 8er Macherey-Nagel flach': 12, '96 Wel

wellSelectionStr (*wells: (<class 'int'>, [<class 'int'>], [<class 'labware.Well'>]) = None*)

> Returns See A.15.3, pag. A-122

file:///C:/Prog/RobotEvo/FreedomEVOwareStandardV2.4SP1-2011.ExtendedDeviceSupportManual.pdf
Many of the advanced worklist commands have a parameter called wellSelection. wellSelection is a
string which specifies the wells (tips) which should be used for the command. Characters 1 and 2 of
the string specify the number of wells in the x-direction in hexadecimal. Characters 3 and 4 of the the
string specify the number of wells in the y-direction in hexadecimal. For example, 12 x 8 (96 wells) =
0C08. All following characters are used for the well selection, whereby each character specifies the well
selection for a group of 7 adjacent wells using a specially adapted bitmap system. Only 7 bits are used
per byte [RANGE 0-127 !!!] instead of 8 to avoid screen and printer font compatibility problems. Using
the 7-bit system, 14 characters are needed to represent the well selection for 96 wells (plus characters 1 to
4, total of 18 characters) and 55 characters are needed to represent the well selection for 384 wells (total
of 59 characters). In addition, since most ANSI characters below ANSI 32 are non-printable (nonhuman-
readable), decimal 48 (ANSI value for "0") is added to the value [RANGE 48-175 !!! 144 have undefined
Unicode !!!] of the bitmap to make it easier to read, send by eMail etc. The following shows some
examples for character 5 of the well selection string for a 96-well microplate in landcape orientation.
Character 5 is responsible for the first group of 7 wells

this function stores 7 bit per character in the selection string the first 2 characters are the number of wells
in x direction (columns) in hexadecimal. the characters 3 and 4 are the number of wells in y direction
(rows) in hexadecimal. well are computed in the order back to front, left to right; https://docs.python.org/
3.4/library/string.html#formatstrings

class labware.**LiquidClass**(*name: str*, *liquid_name: str = ''*)
> Bases: `object`

> **__init__**(*name: str*, *liquid_name: str = ''*)

>> **Parameters**

>>> • **name** –

>>> • **liquid_name** –

class labware.**LiquidClassDefault**(*name: str*, *liquid_name: str = ''*)
> Bases: *labware.LiquidClass*

> **__init__**(*name: str*, *liquid_name: str = ''*)

>> **Parameters**

>>> • **name** –

>>> • **liquid_name** –

class labware.**LiquidClassDerived**(*raw_name: str*, *origen: labware.LiquidClassDefault*)
> Bases: *labware.LiquidClass*

> **__init__**(*raw_name: str*, *origen: labware.LiquidClassDefault*)

>> **Parameters**

>>> • **name** –

>>> • **liquid_name** –

class labware.**LiquidClasses**(*database: pathlib.Path*)
> Bases: `object`

> **__init__**(*database: pathlib.Path*)
> Initialize self. See help(type(self)) for accurate signature.

exception labware.**NoFreeWells**(*labware: labware.Labware*, *error: str*)
> Bases: `Exception`

__init__(*labware: labware.Labware*, *error: str*)
    Initialize self. See help(type(self)) for accurate signature.

exception labware.**ProtocolLogicPippetingError**
    Bases: Exception

class labware.**Te_Mag**(*name*, *nRow*, *nCol=1*, *max_vol=None*)
    Bases: labware.Type

class labware.**Tip**(*rack_type*)
    Bases: object

    __init__(*rack_type*)
        Initialize self. See help(type(self)) for accurate signature.

class labware.**Well**(*labware*, *Well_Offset*)
    Bases: object

    class Action(*volume: float*, *origin=None*)
        Bases: object

        __init__(*volume: float*, *origin=None*)
            Initialize self. See help(type(self)) for accurate signature.

    __init__(*labware*, *Well_Offset*)
        Initialize self. See help(type(self)) for accurate signature.

    actions

    log(*vol*, *origin=None*)

    reagent

    select(*sel=True*)

    vol

class labware.**WorkTable**(*template_file*, *robot_protocol=None*, *grids=67*, *sites=127*)
    Bases: object

    Collection of carriers.types and Labware.types and pos of instances

    class File(*input*, *output*, *worktable*)
        Bases: object

        __init__(*input*, *output*, *worktable*)
            Initialize self. See help(type(self)) for accurate signature.

        grid_carrier_line()

        write(*worktable*)

    class Location(*grid=None*, *site=None*, *carrier=None*, *carrier_site=None*, *worktable=None*)
        Bases: object

        One location in a WorkTable

        __init__(*grid=None*, *site=None*, *carrier=None*, *carrier_site=None*, *worktable=None*)
                Parameters
                    • **grid** – int, 1-67. worktable grid. Carrier grid position
                    • **site** – int, 0 - 127. Site on carrier (on RAck?) = labware location - (site on carrier - 1) !!!!!
                    • **carrier** –
                    • **carrier_site** –

__init__(*template_file*, *robot_protocol=None*, *grids=67*, *sites=127*)
  Initialize self. See help(type(self)) for accurate signature.

add_labware(*labware*, *loc: labware.WorkTable.Location*)

  > **Parameters**
  >
  > > • **labware** –
  > >
  > > • **loc** –
  >
  > **Returns**
  >
  > **Raises** `"This WT have only " + len(self.grid) + " grid."` –

add_new_labware(*labware*, *loc: labware.WorkTable.Location = None*)
  This will be the first location of this labware. Don't remove from possible old location. :param labware:
  :param loc: :return: :raise "This WT have only " + len(self.grid) + " grid.":

cur_worktable = None

get_DITI_series(*rack=None*)
  :type rack:(str, DITIrackType, DITIrack, DITIrackTypeSeries)

get_current_labware(*labware*)

get_labware(*label: (<class 'str'>, <class 'int'>) = None*, *labw_type=None*)
  Return a *Labware* already created manually or after the worktable template was scanned. The labware
  type is optional (if you provide a label), but it makes the search more robust. It is mandatory if you provide
  no label or an index (no label will return the labware with index *0* in the series of labware of the desired
  type). The type may be a label or a predefined *Labware.Type* :type labw_type: (str, Labware.Type) :param
  label: :return Labware

labware_series = None
  typeName: Series. For each type - a series of labwares (with self have locations)

parse_worktable_file(*template_file*, *robot_protocol*)

reagents = None
  connect each reagent name with the reagent self

replace_with_new(*labw*, *label*)

retire_labware(*labw*)

set_current(*labware*)

set_def_DiTi(*tips*)

set_first_pos(*labw_type_name=None*, *posstr=None*)
  Default to DITI if no labw_type_name is given. chooses a labware by label and set next well or tip to be
  used. :param labw_type_name: :param posstr: :return:

class labware.**conectedWell**(*labware*, *Well_Offset*)
  Bases: *labware.Well*

  actions

  reagent

  vol

labware.**count_tips**(*TIP_MASK: int*) → int

labware.**getLabware**(*labw_type*, *label*, *worktable=None*)

class labware.**usedTip**(*tip: labware.Tip*, *origin=None*)
    Bases: *labware.Tip*

    __init__(*tip: labware.Tip*, *origin=None*)
        Initialize self. See help(type(self)) for accurate signature.

class instructions.**RoMa**(*action: int*, *distance: float*, *force: int*, *xOffset: float*, *yOffset: float*, *zOffset: float*, *xyzSpeed: float*, *xyzMax: int*, *romaNo: int*)
    Bases: EvoScriPy.Instruction_Base.Instruction

15.60 Move RoMa Command This command is used to carry out simple RoMa movements without using a RoMa vector: The parameters of the Move RoMa command are as follows:

    • ROMA-No.

Choose the RoMa you want to use (1 or 2) if your instrument is fitted with more than one.

    • Open Gripper

This opens the gripper to the specified width (range: 55 to 140 mm). Close Gripper

This closes the gripper to the specified width (range: 55 to 140 mm) using the specified force (range: 0 to 249). A Grip Error message will be output if no resistance is detected when gripping.

    • Move to home position

This moves the RoMa to the specified home position. After completing a sequence of movements with the RoMa, you should move it back to its home (parking) position, out of the way of other objects on the worktable (see 9.6.4 "Defining the Home Position for a RoMa", 9-63).

    • Move relative to current position

This moves the RoMa relative to its current position. You must then specify the relative distances (range: -400 to 400 mm) and whether the RoMa should move at a particular speed (range: 0.1 to 400 mm/s) or at maximum speed. The movements in A and B are not aligned with the instrument axes, but with the current rotator position (angle). The movement in A is perpendicular to the gripper. Example #1: If the gripper points to the front (angle = 0°) and the A value is positive, then the RoMA will move to the left. Example #2: If the gripper points to the left (angle = 90°) and the A value is positive, then the RoMA will move to the back. The movement in B is in line with the gripper. Example #1: If the gripper points to the front (angle = 0°) and the B value is positive, then the RoMA will move to the front. Example #2: If the gripper points to the left (angle = 90°) and the B value is positive, then the RoMA will move to the left. See also 9.4.8.2 "RoMa Coordinate System", 9-39.

    RoMa_1 = 0

    RoMa_2 = 1

    __init__(*action: int*, *distance: float*, *force: int*, *xOffset: float*, *yOffset: float*, *zOffset: float*, *xyzSpeed: float*, *xyzMax: int*, *romaNo: int*)

        Parameters

            • **action** – 0 = open gripper, 1 = close, 2 = move to home position, 3 = move relative to current position

            • **distance** – 55 - 140, gripper distance in mm for opening or closing

            • **force** – 0 - 249, force when closing gripper

            • **xOffset** – -400 - 400, x-distance for relative move in mm

            • **yOffset** (*object*) – -400 - 400, y-distance for relative move in mm

            • **zOffset** – -400 - 400, z-distance for relative move in mm

            • **xyzSpeed** – 0.1 - 400, speed in mm/s

- **xyzMax** – 0 = use xyzSpeed, 1 = use maximum speed

- **romaNo** – number of the RoMa performing the action: 0 = RoMa 1, 1 = RoMa 2

actualize_robot_state()

close_gripper = 1

home_position = 3

move_relative = 4

move_to = 2

open_gripper = 0

use_max_speed = 1

use_xyzSpeed = 0

validate_arg()

class instructions.**activate_PMP**(*tipMask=None*)
Bases: EvoScriPy.Instruction_Base.Instruction

A.15.4.12 Activate PMP (Worklist: Activate_PMP)

__init__(*tipMask=None*)
Initialize self. See help(type(self)) for accurate signature.

exec(*mode=None*)

validate_arg()

class instructions.**active_Wash**(*wait=True*, *time=None*, *arm=None*)
Bases: EvoScriPy.Instruction_Base.Instruction

A.15.4.16 Active WashStation (Worklist: Active_Wash)

__init__(*wait=True*, *time=None*, *arm=None*)
Initialize self. See help(type(self)) for accurate signature.

validate_arg()

class instructions.**aspirate**(*tipMask=None*, *liquidClass=None*, *volume=None*, *labware=None*,
spacing=1, *wellSelection=None*, *LoopOptions=[]*, *Rack-
Name=None*, *Well=None*, *arm=None*)
Bases: EvoScriPy.Instruction_Base.Pipetting

A.15.4.1 Aspirate command (Worklist: Aspirate) A - 125

__init__(*tipMask=None*, *liquidClass=None*, *volume=None*, *labware=None*, *spacing=1*, *wellSelec-
tion=None*, *LoopOptions=[]*, *RackName=None*, *Well=None*, *arm=None*)

Parameters

- **liquidClass** –

- **volume** –

- **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

- **labware** – Labware;

- **spacing** – int; tip spacing

- **wellSelection** – str;

- **LoopOptions** – list; of objects of class LoopOption.

- **RackName** –

- **Well** –

- **arm** –

static action()

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**comment**(*text*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.21 Comment (Worklist: Comment)

> **__init__**(*text*)

> **validate_arg()**

class instructions.**deactivate_PMP**(*tipMask=None*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.13 Deactivate PMP (Worklist: Deactivate_PMP)

> **__init__**(*tipMask=None*)
> > Initialize self. See help(type(self)) for accurate signature.

> **exec**(*mode=None*)

> **validate_arg()**

class instructions.**detect_Liquid**(*tipMask=None*, *liquidClass=None*, *labware=None*, *spacing=1*, *wellSelection=None*, *LoopOptions=None*, *arm=None*, *RackName=None*, *Well=None*)
> Bases: EvoScriPy.Instruction_Base.Pipetting

> A.15.4.11 Detect Liquid (Worklist: Detect_Liquid) Liquid level detection is one of the options available for aspirating and dispensing and can be individually defined for each liquid class. The Detect Liquid command is used to carry out liquid level detection without pipetting and reports the liquid volume for each of the chosen wells in the labware. The volumes are returned in a set of variables DETECTED_VOLUME_x, where x is the tip number.

> **__init__**(*tipMask=None*, *liquidClass=None*, *labware=None*, *spacing=1*, *wellSelection=None*, *LoopOptions=None*, *arm=None*, *RackName=None*, *Well=None*)
> > Set labware to match wells.

> > **Parameters**

> > - **name** – str; Instruction name

> > - **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

> > - **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

> > - **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip

> spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

> - **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.
>
> - **LoopOptions** – list; of objects of class LoopOption.
>
> - **RackName** –
>
> - **Well** –
>
> - **arm** –

> **static action**()

class instructions.**dispense**(*tipMask=None, liquidClass=None, volume=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], labware=None, spacing=1, wellSelection=None, LoopOptions=[], RackName=None, Well=None, arm=None*)

> Bases: EvoScriPy.Instruction_Base.Pipetting

> A.15.4.2 Dispense (Worklist: Dispense)

> **__init__**(*tipMask=None, liquidClass=None, volume=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], labware=None, spacing=1, wellSelection=None, LoopOptions=[], RackName=None, Well=None, arm=None*)
> > Set labware to match wells.

> > **Parameters**

> > - **name** – str; Instruction name
> >
> > - **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)
> >
> > - **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells
> >
> > - **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).
> >
> > - **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.
> >
> > - **LoopOptions** – list; of objects of class LoopOption.
> >
> > - **RackName** –
> >
> > - **Well** –
> >
> > - **arm** –

> **static action**()

> **validate_arg**()
> > Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**dropDITI**(*tipMask=None, labware=None, AirgapVolume=0, AirgapSpeed=300, arm=None*)

> Bases: EvoScriPy.Instruction_Base.Pipette

A.15.4.6 Drop DITIs command (Worklist: DropDITI). pag A - 130 and 15 - 14

__init__(*tipMask=None*, *labware=None*, *AirgapVolume=0*, *AirgapSpeed=300*, *arm=None*)

> Parameters

> > • **conditional** – exec only if there are some tip to droop.

> > • **tipMask** –

> > • **labware** – Specify the worktable position for the DITI waste you want to use. You must first put a DITI waste in the Worktable at the required position.

> > • **AirgapVolume** – floating point, 0 - 100. airgap in μl which is aspirated after dropping the DITIs

> > • **AirgapSpeed** – int 1-1000. Speed for the airgap in μl/s

> > • **arm** –

actualize_robot_state()

exec(*mode=None*)

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**execute**(*application*, *options*, *responseVariable*, *scope=0*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.20 Execute Application (Worklist: Execute)

> **__init__**(*application*, *options*, *responseVariable*, *scope=0*)

> **validate_arg**()

class instructions.**execute_VBscript**(*filename*, *action=0*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.24 Execute VB Script (Worklist: Execute_VBscript)

> **__init__**(*filename*, *action=0*)

> > Parameters

> > > • **filename** – Path and filename of the defined VB script.

> > > • **action** – Use Waits, Continues and Waits_previous defined in subroutine

> **validate_arg**()

class instructions.**export**(*exportAll=True*, *formats=32*, *delete=False*, *compress=False*, *Raks=None*, *significantStep=1*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.17 Export Data (Worklist: Export)

> **__init__**(*exportAll=True*, *formats=32*, *delete=False*, *compress=False*, *Raks=None*, *significantStep=1*)
> > Initialize self. See help(type(self)) for accurate signature.

> **dbase = 2**

> **excel = 4**

> **lotus = 1**

> **paradox = 8**

```
        quattro = 16

        text_with_delimiters = 32

        validate_arg()
```

class instructions.**getDITI**(*tipMask*, *type*, *options=0*, *arm=None*)

> Bases: EvoScriPy.Instruction_Base.DITIs

> **__init__**(*tipMask*, *type*, *options=0*, *arm=None*)

>> **A.15.4.5 Get DITIs (Worklist: GetDITI) ...** The Get DITIs command is used to pick up DITIs (disposable tips) of the specified type from a DITI rack. Freedom EVOware keeps track of their position on the worktable and automatically picks up the next available unused DITIs of the chosen type. When you choose a DITI type in a script command, the pull-down list all of the LiHa DITI types which are currently configured in the labware database. When you want to pick up a DiTi, Freedom EVOware searches the worktable for a DITI rack which contains the DITI type you have specified in the script command. To configure Freedom EVOware for a new DITI type, create a new DITI rack or duplicate an existing DITI rack and give the new labware a suitable name (e.g. "ZipTip"). DiTi Index: Freedom EVOware automatically assigns a unique numeric index to each DITI type. You cannot edit the index manually. The DITI index is used e.g. by the Set DITI Type command in worklists and in advanced worklists. The DITI index is shown in the Edit Labware dialog box for the DITI labware (Well Dimensions tab). This function is deprecated in favor of getDITI2 which do not use index Currently only use ... ?

>> **Parameters**

>>> • **label** –

>>> • **tipMask** –

>>> • **type** – int, 0-3. DITI index (see 9.4.5 "Labware Types and DITI Types", 9-32, DITI Index).

> **validate_arg**()

class instructions.**getDITI2**(*tipMask=None*, *DITI_series:* *(<class* *'str'>*, *<class* *'EvoScriPy.labware.DITIrackType'>*, *<class* *'EvoScriPy.labware.DITIrack'>*, *<class* *'EvoScriPy.labware.DITIrackTypeSeries'>)* *=* *None*, *options=0*, *arm=None*, *AirgapVolume=0*, *AirgapSpeed=300*)

> Bases: EvoScriPy.Instruction_Base.DITIs

> A.15.4.5 Get DITIs (Worklist: GetDITI) pag. A - 129 It take a labware type or name instead of the labware itself because the real robot take track of the next position to pick including the rack and the site (that is - the labware). It need a labware type and it know where to pick the next tip.

> **__init__**(*tipMask=None*, *DITI_series:* *(<class 'str'>*, *<class 'EvoScriPy.labware.DITIrackType'>*, *<class 'EvoScriPy.labware.DITIrack'>*, *<class 'EvoScriPy.labware.DITIrackTypeSeries'>)* *= None*, *options=0*, *arm=None*, *AirgapVolume=0*, *AirgapSpeed=300*)

>> **Parameters**

>>> • **tipMask** –

>>> • **DITI_series** – string or labware or **labware_**.Type? DiTi labware name

>>> • **options** –

>>> • **arm** –

>>> • **AirgapVolume** – int. used to specify a system trailing airgap (STAG) which will be aspirated after mounting the DITIs. Volume in μl

- **AirgapSpeed** – int. Speed for the airgap in μl/s

actualize_robot_state()

exec(*mode=None*)

validate_arg()

class instructions.**group**(*titel*)
Bases: EvoScriPy.Instruction_Base.ScriptONLY

UNDOCUMENTED. Begging a group. MANUALLY set the group_end() !!!!

__init__(*titel*)

validate_arg()

class instructions.**group_end**
Bases: EvoScriPy.Instruction_Base.ScriptONLY

UNDOCUMENTED. Begging a group. MANUALLY set the group_end() !!!!

__init__()

validate_arg()

class instructions.**mix**(*tipMask=None, liquidClass=None, volume=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], labware=None, spacing=1, wellSelection=None, cycles=3, LoopOptions=[], RackName=None, Well=None, arm=None*)
Bases: EvoScriPy.Instruction_Base.Pipetting

A.15.4.3 Mix (Worklist: Mix)

__init__(*tipMask=None, liquidClass=None, volume=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], labware=None, spacing=1, wellSelection=None, cycles=3, LoopOptions=[], RackName=None, Well=None, arm=None*)
Set labware to match wells.

Parameters

- **name** – str; Instruction name

- **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

- **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

- **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

- **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

- **LoopOptions** – list; of objects of class LoopOption.

- **RackName** –

- **Well** –

> • **arm** –

static action()

actualize_robot_state()

validate_arg()

>   Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**moveLiha**(*zMove*, *zTarget*, *offset*, *speed*, *tipMask=None*, *labware=None*, *spacing=1*, *wellSelection=None*, *LoopOptions=[]*, *RackName=None*, *Well=None*, *arm=None*)

Bases: EvoScriPy.Instruction_Base.Pipette

A.15.4.14 Move LiHa (Worklist: MoveLiha - A - 135)see 15.21 "Move LiHa Command", 15-33. The Move LiHa command is used to move the liquid handling arm (LiHa) from one position to another without performing an Aspirate or Dispense operation. Type of movement Choose X-Move, Y-Move or Z-Move to move only one axis of the LiHa. You can then specify the speed of the movement. Z-Move only moves the selected tips. The options Positioning with global Z-travel, Positioning with local Z-travel and Positioning with variable Z-travel move the LiHa to the labware at maximum speed. The chosen height for Z-Travel (the tip height which is used during the arm movement) only applies to the selected tips. The Z position of the unselected tips remains unchanged. If you choose Positioning with variable Z-travel, the required Z-Travel height is specified using the pre-defined variable LIHA_MOVE_HEIGHT (see 14.1.4.7 "LIHA_MOVE_HEIGHT", 14-5). Z-Position Unless you have chosen X-Move or Y-Move in the Type of Movement field, you can specify the Z-Position to which the selected tips should be lowered at the end of the LiHa movement. The Z position of the unselected tips remains unchanged. Choose the required Z-position and then specify a Z offset in mm if required. A positive value for the offset lowers the tips.

__init__(*zMove*, *zTarget*, *offset*, *speed*, *tipMask=None*, *labware=None*, *spacing=1*, *wellSelection=None*, *LoopOptions=[]*, *RackName=None*, *Well=None*, *arm=None*)

>   **Parameters**
>
>   - **zMove** – int; type of movement: 0 = positioning with global z-travel 1 = positioning with local z-travel 2 = x-move 3 = y-move 4 = z-move
>
>   - **zTarget** – int; z-position after move: 0 = z-travel 1 = z-dispense 2 = z-start 3 = z-max 4 = global z-travel
>
>   - **offset** – float; in range (-1000, 1000) offset in mm added to z-position (parameter z_target)
>
>   - **speed** – float; in range (0.1, 400) move speed in mm/s if z_move is x-move, y-move or z-move
>
>   - **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)
>
>   - **labware** – Labware;
>
>   - **spacing** – int; tip spacing
>
>   - **wellSelection** – str; bit-coded well selection
>
>   - **LoopOptions** – list; of objects of class LoopOption.
>
>   - **RackName** –
>
>   - **Well** –
>
>   - **arm** –

global_z_travel = 4

pos_global_z_travel = 0

pos_local_z_travel = 1

validate_arg()
>   Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

x_move = 2

y_move = 3

z_dispense = 1

z_max = 3

z_move = 4

z_start = 2

z_travel = 0

class instructions.**notification**(*receiverGroup*, *AttachScreen_ShotFlag=False*, *emailSubject="*, *emailMessage="*, *action=0*)
>   Bases: EvoScriPy.Instruction_Base.Instruction

>   A.15.4.25 Notification (Worklist: Notification)

>   **__init__**(*receiverGroup*, *AttachScreen_ShotFlag=False*, *emailSubject="*, *emailMessage="*, *action=0*)

>   >   **Parameters**

>   >   >   • **receiverGroup** –

>   >   >   • **AttachScreen_ShotFlag** –

>   >   >   • **emailSubject** –

>   >   >   • **action** – 0 = send email now, 1 = send email on error, 2 = stop sending email on error

>   **validate_arg**()

class instructions.**pickUp_DITIs**(*tipMask=None*, *labware=None*, *wellSelection=None*, *LoopOptions=[]*, *type=None*, *arm=None*, *RackName=None*, *Well=None*)
>   Bases: EvoScriPy.Instruction_Base.Pipette

>   A.15.4.8 Pick Up DITIs (Worklist: Pick Up_DITI) pag. A-131 and 15-16 The Pick Up DITIs command is used to pick up DITIs which have already been used and put back into a DITI rack with the Set DITIs Back command. You must specify the DITIs you want to pick up.

>   **__init__**(*tipMask=None*, *labware=None*, *wellSelection=None*, *LoopOptions=[]*, *type=None*, *arm=None*, *RackName=None*, *Well=None*)
>   >   Set labware to match wells.

>   >   **Parameters**

>   >   >   • **name** – str; Instruction name

>   >   >   • **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

>   >   >   • **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

>   >   >   • **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other

well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

- **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

- **LoopOptions** – list; of objects of class LoopOption.

- **RackName** –

- **Well** –

- **arm** –

actualize_robot_state()

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**pickUp_DITIs2**(*tipMask=None*,   *labware=None*,   *wellSelection=None*, *LoopOptions=[], arm=None, RackName=None, Well=None*)
> Bases: EvoScriPy.Instruction_Base.Pipette

A.15.4.8 Pick Up DITIs (Worklist: Pick Up_DITI) pag. A-131 and 15-16 NOT DOCUMENTED

The Pick Up DITIs command is used to pick up DITIs which have already been used and put back into a DITI rack with the Set DITIs Back command. You must specify the DITIs you want to pick up.

__init__(*tipMask=None*, *labware=None*, *wellSelection=None*, *LoopOptions=[]*, *arm=None*, *RackName=None*, *Well=None*)
> Set labware to match wells.

> **Parameters**

> - **name** – str; Instruction name

> - **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

> - **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

> - **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

> - **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

> - **LoopOptions** – list; of objects of class LoopOption.

> - **RackName** –

> - **Well** –

> - **arm** –

actualize_robot_state()

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**pickUp_ZipTip**(*tipMask=None*)
> Bases: EvoScriPy.Instruction_Base.Pipette

> A.15.4.10 Pickup ZipTip (Worklist: PickUp_ZipTip)

> **__init__**(*tipMask=None*)
> > Set labware to match wells.

> > **Parameters**

> > > • **name** – str; Instruction name

> > > • **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

> > > • **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

> > > • **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

> > > • **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

> > > • **LoopOptions** – list; of objects of class LoopOption.

> > > • **RackName** –

> > > • **Well** –

> > > • **arm** –

class instructions.**set_DITI_Counter**(*type=None*, *posInRack=0*, *labware=None*)
> Bases: EvoScriPy.Instruction_Base.Pipette

> **A.15.4.7 Set Diti Position (Worklist: Set_DITI_Counter) pag. 15 - 15** If you are using DITIs, Freedom EVOware remembers the position in the DITI

rack of the last DITI which was fetched. When starting a new run, the Get DITIs command starts picking up DITIs at the next available position. After loading a new DITI rack onto the worktable during script runtime (e.g. using the RoMa), you should use the Set DITI Position command in your script to set the DITI Position counter to 1. This ensures that the next DITI is fetched from position 1 rather than from the middle of the new rack. You can specify the next position separately for each of the available DITI types (i.e. DITI racks on the worktable). Note: If you want to specify the next DITI position manually before the script or process is started, use the direct command Set DITI Position (see 5.4.1.3 "Direct commands", 5-10) or create a maintenance script which contains the Set DITI Position command (see 6.4.2 "Run Maintenance", 6-10). Note: DiTi handling is automatic in Freedom EVOware Plus. This command is only shown in the Control Bar if you are using DiTis on the LiHa. Freedom EVOware does not detect the LiHa tip type automatically. If you are using DITIs you must configure them manually (see 8.4.2.1 "LiHa (Liquid Handling Arm)", 8-22). If your pipetting instrument is fitted with two liquid handling arms, the Set DITI Position command will be provided in the Control Bar for

both arms. However, please note that the same DITI position counter (and the same pool of unused DITIs) is used by both arms.

__init__ (*type=None, posInRack=0, labware=None*)
   Set labware to match wells.

> **Parameters**
>
> > - **name** – str; Instruction name
> >
> > - **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)
> >
> > - **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells
> >
> > - **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).
> >
> > - **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.
> >
> > - **LoopOptions** – list; of objects of class LoopOption.
> >
> > - **RackName** –
> >
> > - **Well** –
> >
> > - **arm** –

actualize_robot_state ()

validate_arg ()
   Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**set_DITI_Counter2** (*labware=None, posInRack=0, lastPos=False*)
   Bases: EvoScriPy.Instruction_Base.Pipette

   A.15.4.7 Set Diti Position (Worklist: Set_DITI_Counter) NOT DOCUMENTED example:
   > Set_DITI_Counter2("DiTi 1000ul","25","2","5",0); last position

   If you have activated the feature Optimize positions when fetching DITIs, Freedom EVOware fetches new DITIs either starting from the beginning of the DITI rack or starting from the end of the DITI rack, depending on the situation (see 8.4.2.1 "LiHa (Liquid Handling Arm)", 8-22, Optimize positions when fetching DITIs). In this case, Freedom EVOware maintains two counters for the last used DITI position (for DITIs which are taken from the beginning of the rack and for DITIs which are taken from the end of the rack). Check this checkbox if you want to set the last used DITI position for the end counter instead of for the beginning counter. If you have activated the feature Optimize positions when fetching DITIs, after loading a new DITI rack onto the worktable during script runtime you should use the Set DITI Position command twice in your script, to set the beginning counter to 1 and the end counter to 96. The Set last position checkbox is inactive (grey) if you have not activated Optimize positions when fetching DITIs. If you have previously specified the last used DITI position, it will be ignored during script execution

__init__ (*labware=None, posInRack=0, lastPos=False*)
   Set labware to match wells.

Parameters

- **name** – str; Instruction name

- **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

- **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

- **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates, you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

- **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

- **LoopOptions** – list; of objects of class LoopOption.

- **RackName** –

- **Well** –

- **arm** –

actualize_robot_state()

validate_arg()

Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**set_DITIs_Back**(*tipMask*, *labware: EvoScriPy.labware.DITIrack*, *wellSelection=None*, *LoopOptions=[]*, *arm=None*, *RackName=None*, *Well=None*)
    Bases: EvoScriPy.Instruction_Base.Pipette

A.15.4.9 Set DITIs Back (Worklist: Set_DITIs_Back) return used DITIs to specified positions on a DITI rack for later use. This command requires the Lower DITI Eject option.

__init__(*tipMask*, *labware: EvoScriPy.labware.DITIrack*, *wellSelection=None*, *LoopOptions=[]*, *arm=None*, *RackName=None*, *Well=None*)
    Set labware to match wells.

Parameters

- **name** – str; Instruction name

- **tipMask** – int; selected tips, bit-coded (tip1 = 1, tip8 = 128)

- **labware** – Labware; grid 1-67, site 0-127, the labware with the selected wells

- **spacing** – int; Tip Spacing The Tip Spacing parameter controls the distance between adjacent pipetting tips for this command. You can choose a different tip spacing for the source labware and the destination labware. Tip spacing is only relevant if you want to use more than one tip. A tip spacing of 1 means that the tips will be spread to match the distance between adjacent wells in the labware. A tip spacing of 2 will select every other well in the labware. You can only choose values for tip spacing which are meaningful for the labware geometry. The liquid handling arm achieves the highest mechanical accuracy when the tips are not spread. For high-density labware such as 1536-well microplates,

you should choose tip spacing such that the tips are adjacent to one another (physical tip spacing 9 mm). Accordingly, for 1536-well microplates you should set tip spacing to 4 (every fourth well).

- **wellSelection** – str; list of wells. Converted to bit-coded well selection to be used.

- **LoopOptions** – list; of objects of class LoopOption.

- **RackName** –

- **Well** –

- **arm** –

actualize_robot_state()

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**startTimer**(*timer=1*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.18 Start Timer (Worklist: StartTimer)

> **__init__**(*timer=1*)

>> **Parameters timer** (*Expression*) – expression, 1 - 100. number of timer to re-start. 1-1000?

> **validate_arg**()

class instructions.**subroutine**(*filename*, *action=0*)
> Bases: EvoScriPy.Instruction_Base.ScriptONLY

> UNDOCUMENTED

> **Continues = 1**

> **Waits = 0**

> **Waits_previous = 2**

> **__init__**(*filename*, *action=0*)

> **validate_arg**()

class instructions.**transfer_rack**(*labware: EvoScriPy.labware.Labware*, *destination: EvoScriPy.labware.WorkTable.Location*, *vectorName: str = None*, *backHome: bool = True*, *slow: bool = True*, *lid: EvoScriPy.labware.Labware = None*, *cover: int = 0*, *romaNo: int = None*)
> Bases: EvoScriPy.Instruction_Base.Instruction

This command is used to transfer labware (e.g. a microplate) from one position to another with the plate robot (RoMa). If you have scanned the labware barcode and you move the labware with the Transfer Labware command, the barcode remains assigned to the labware (i.e. the labware data record) at the new location. In addition, pipetting information, if any, remains assigned to the labware (see 15.29 "Export Data Command", 15- 50). Grip and release commands for the RoMa (used to pick up and put down the labware) are handled automatically. The required gripper spacing is taken from the advanced properties for the respective labware type (see 9.4.2 "Editing Labware, Advanced Tab", 9-22): The parameters of the Transfer Labware command are as follows:

> Move with

Choose the RoMa you want to use (1 or 2) if your instrument is fitted with more than one.

Vector

Choose the RoMa vector that you want to use to pick up the labware. Choose Narrow to pick up the labware on the narrow side; choose Wide to pick up the labware on the wide side. Choose User defined (Narrow) or (Wide) to pick up the labware on the narrow or wide side with a user-defined vector. In this case, you must choose the user-defined vector to use for picking up the labware at the source position and putting down it at the destination position in the two User Vector pull-down lists. User-defined vectors are created in the Control Bar (Robot Vectors section). See 5.4.1.4 "Robot Vectors", 5-11. Above all if you did not create the user-defined vector yourself, we recommend you to check carefully that the vector moves the RoMa to the correct (i.e. intended) source and destination carrier positions before using it for pipetting. Tip: Test the RoMa vector in a script using the 3D simulation program EVOSim. Freedom EVOware will report an error when you complete the Transfer Labware command if the narrow or wide RoMa vector has not yet been created for the chosen labware type. It is best to create the required RoMa vectors in advance (see 9.6.2 "Teach Plate Robot Vector Dialog Box", 9-60).

Transfer Labware command, Step 1

Specify the parameters for the source position:

- Source position

Select the current position of the labware by clicking on it in the Worktable Editor. Grid and Site then show the position you have chosen. The gray (protected) field Defined Carrier then shows the type of carrier at the chosen site and the gray (protected) field Defined Labware shows the type of labware at the chosen site. If you want to fetch the labware from a device such as a hotel or barcode scanner, click on the device icon. You then need to choose the site and the labware type. The list shows labware types which are allowed for the device (see 9.5 "Configuring Carriers", 9-39, "Allowed Labware on this carrier").

Transfer Labware command, Step 2

Specify the parameters for the labware lid. These parameters are only available for labware types which can be fitted with a lid:

- Lid handling

Check this checkbox if the labware has a lid. Choose Cover at source if you want to put on the lid before moving the labware. Choose Uncover at destination if you want to remove the lid after moving the labware to the destination position (i.e. the lid was already present). In either case, select the position for fetching or putting aside the lid by clicking on the site in the Worktable Editor. Grid and Site then show the position you have chosen. The gray (protected) field Defined Carrier shows the type of carrier on which the lid is placed/will be placed. You can only put aside lids on unused carrier sites.

Transfer Labware command, Step 3

Specify the parameters for the destination position:

- Destination Position

Select the required destination position of the labware by clicking on the site in the Worktable Editor. The destination site must be suitable for the labware type you are moving (see 9.5 "Configuring Carriers", 9-39, "Allowed labware on this carrier"). Grid and Site then show the position you have chosen. The gray (protected) field Defined Carrier then shows the type of carrier at the chosen site. If you want to move the labware to a device such as a hotel or barcode scanner, click on the device icon. You then need to choose the site and the labware type. The list shows labware types which are allowed for the device (see 9.5 "Configuring Carriers", 9-39, "Allowed Labware on this carrier").

- Speed

Choose Maximum if you want the RoMa to move at maximum speed. Choose Taught in vector dialog if you want the RoMa to move at the speed specified in the RoMa vector.

> • Move back to Home Position

Check this checkbox if you want the RoMa to move back to its home (parking) position after transferring the labware. See 9.6.4 "Defining the Home Position for a RoMa", 9-63.

> **__init__**(*labware:* *EvoScriPy.labware.Labware*, *destination:* *Evo-*
> *ScriPy.labware.WorkTable.Location*, *vectorName:* *str = None*, *backHome:* *bool =*
> *True*, *slow: bool = True*, *lid: EvoScriPy.labware.Labware = None*, *cover: int = 0*, *romaNo:*
> *int = None*)
>
>> **Parameters**
>>
>>> • **labware** –
>>>
>>> • **destination** –
>>>
>>> • **backHome** – move back to home when finished ?
>>>
>>> • **lid** –
>>>
>>> • **slow** – use slow speed (as defined in RoMa vector)? (else use maximum speed)
>>>
>>> • **cover** – 0 = cover at source , 1 = uncover at destination
>>>
>>> • **vectorName** – name of RoMa vector to use (as in the Freedom EVOware configuration), choose from one of the following:
>>>
>>>> Narrow DriveIN_Narrow DriveIN_Narrow DriveIN_Wide
>>>
>>> • **romaNo** – number of the RoMa performing the action: 0 = RoMa 1, 1 = RoMa 2

> **actualize_robot_state**()

> **validate_arg**()

class instructions.**userPrompt**(*text: str*, *sound: int = 1*, *closeTime: int = -1*)

> Bases: `EvoScriPy.Instruction_Base.Instruction`

> A.15.4.22 User Prompt (Worklist: UserPrompt)

> **__init__**(*text: str*, *sound: int = 1*, *closeTime: int = -1*)

> **validate_arg**()

class instructions.**variable**(*var_name*, *default*, *queryFlag=False*, *queryString="*, *checkLim-*
> *its=False*, *lowerLimit=0.0*, *upperLimit=0.0*, *type=0*, *scope=0*, *Init-*
> *Mode=0*, *QueryAtStart=False*)
> Bases: `EvoScriPy.Instruction_Base.Instruction`

> A.15.4.23 Set Variable (Worklist: Variable)

> **File_import = 2**

> **Fixed_value = 0**

> **Instance = 1**

> **Numeric = 0**

> **Run = 0**

> **Script = 2**

> **String = 1**

> **User_query = 1**

> **__init__**(*var_name*, *default*, *queryFlag=False*, *queryString="*, *checkLimits=False*, *lowerLimit=0.0*,
> *upperLimit=0.0*, *type=0*, *scope=0*, *InitMode=0*, *QueryAtStart=False*)

Parameters

- **var_name** – string2 ; name of variable

- **default** – Expression ; value assigned to variable or default value if user query

- **queryFlag** – bool

- **queryString** – String1 ; text shown in user query

- **checkLimits** – bool

- **lowerLimit** –

- **upperLimit** –

- **type** – type of variable; 0 = Numeric; 1 = String

- **scope** – scope of variable (see 6.4.6, 6-12):0 = Run; 1 = Instance; 2 = Script

- **InitMode** – 0 = Fixed value; 1 = User query; 2 = File import;

- **QueryAtStart** – bool ; 1 = Prompt for value at start of script

validate_arg()

class instructions.**vector**(*name*)

Bases: EvoScriPy.Instruction_Base.Instruction

15.61 RoMa Vector Command This command executes a RoMa vector, which is a predefined sequence of RoMa movements. You can also specify gripper actions at the Safe and End positions. See 9.6.1 "Using RoMa Vectors", 9-59 for more information. The RoMa Vector command is intended for special RoMa movements and not for moving labware from one position to another - you should use the Transfer Labware command instead for this purpose. See also 15.61.1 "Moving labware with the RoMa Vector command", 15-145. The parameters of the RoMa Vector command are as follows:

- RoMa-No.

Choose the RoMa you want to use (1 or 2) if your instrument is fitted with more than one.

- Use RoMa Vector

Choose the RoMa vector you want to use for the RoMa movement. The popup list shows the RoMa vectors which are currently defined in the Freedom EVOware database. The digit at the end of the vector name (e.g. Carousel_Narrow_1) indicates the RoMa for which the vector was defined (1 or 2). See also 9.6.1 "Using RoMa Vectors", 9-59. You can also choose a user-defined vector. User-defined vectors are created in the Control Bar (Robot Vectors section). Then specify the grid position and carrier site for which the vector is intended. The Grid field is protected (gray) if you have chosen a vector for a device which is not positioned on the worktable (see Carrier is Device checkbox in the carrier definition). Tip: If you click on a carrier, the current grid position is shown in the small yellow tab at the bottom left.

- Move along RoMa Vector

Choose the required direction of the RoMa movement. Click And back if you want the RoMa to return to the Safe position after reaching the End position.

- Gripper action

Choose the gripper action which should be executed at the Safe position and at the End position. The required gripper spacing to pick up and release the labware is taken from the Grip Width and Release Width parameters in the chosen RoMa vector.

- Speed

Choose Maximum if you want the RoMa to move at maximum speed. Choose Slow if you want the RoMa to move at the speed specified in the RoMa vector.

15.61.1 Moving labware with the RoMa Vector command If you have scanned the labware barcode and you move the labware with the Transfer Labware command, the barcode remains assigned to the labware (i.e. the labware data record) at the new location. In addition, pipetting information, if any, remains assigned to the labware (see 15.29 "Export Data Command", 15- 50). This is not the case with the RoMa Vector command. The barcode and the pipetting information are no longer available at the new location. This also applies analogously to the MCA96 Vector command and the MCA384 Vector command. Proceed as follows if you want to use a Vector command to move barcoded labware in special situations: After scanning the barcode, assign it to a temporary variable (see

14.1.11 "Labware Attributes and String Variables", 14-16).

Move the labware. Re-assign the barcode from the temporary variable to the labware. This workaround transfers the barcode but not the pipetting information.

class `instructions.`**`waitTimer`**(*timer=1*, *timeSpan=None*)
    Bases: `EvoScriPy.Instruction_Base.Instruction`

A.15.4.19 Wait for Timer (Worklist: WaitTimer)

`__init__`(*timer=1*, *timeSpan=None*)

> **Parameters**

>> • **`timeSpan`** – expression, 0.02 - 86400. duration

>> • **`timer`** – expression, 1 - 100. number of timer to re-start. 1-1000?

`validate_arg`()

class `instructions.`**`wash_tips`**(*tipMask=None*, *WashWaste=None*, *WashCleaner=None*, *wasteVol=100*, *wasteDelay=50*, *cleanerVol=10*, *cleanerDelay=50*, *Airgap=0.0*, *airgapSpeed=50*, *retractSpeed=100*, *FastWash=False*, *lowVolume=False*, *atFrequency=0*, *RackName=None*, *Well=None*, *arm=None*)
    Bases: `EvoScriPy.Instruction_Base.Pipette`

A.15.4.4 Wash Tips (Worklist: Wash) pag. A - 128; pag. 15 - 8. to flush and wash fixed tips or to flush DITI

adapters using a wash station. It is not intended for flushing DITI tips (DITI tips should not normally be flushed). Tips should be washed as often as necessary, e.g. after a pipetting sequence and before taking a new sample. DITI adapters should be flushed after replacing the DITIs several times to renew the system liquid column in the DITI adapters. This ensures maximum pipetting accuracy.

`__init__`(*tipMask=None*, *WashWaste=None*, *WashCleaner=None*, *wasteVol=100*, *wasteDelay=50*, *cleanerVol=10*, *cleanerDelay=50*, *Airgap=0.0*, *airgapSpeed=50*, *retractSpeed=100*, *FastWash=False*, *lowVolume=False*, *atFrequency=0*, *RackName=None*, *Well=None*, *arm=None*)

> **Parameters**

>> • **`tipMask`** –

>> • **`WashWaste`** – labware ; the waste you want to use. You must first put a wash station with waste unit in the Worktable Editor at the required position.

>> • **`WashCleaner`** – labware ; the cleaner you want to use. You must first put a wash station with cleaner unit in the Worktable Editor at the required position. Choose a shallow cleaner if you only need to clean the ends of the tips. Choose a deep cleaner if there is a possibility of contamination along the shaft of the tip. The deep cleaner requires a larger volume of

system liquid and cleaning takes somewhat longer. The wash cycle is skipped if you are flushing DITI adapters and Use Cleaner is ignored in this case.

- **wasteVol** – int ; volume [in mL !!] of system liquid which should be used to flush the inside of the tips. Flushing takes place with the tips positioned above the waste of the specified wash station (tip height for fixed tips = Z-dispense; tip height for DITI adapters = Z-travel).

- **wasteDelay** –

- **cleanerVol** – int ; Specify the volume of system liquid which should be used to wash the outside of the tips. Washing takes place with the tips lowered into the cleaner of the specified wash station (tip height = Z-max). The wash cycle is skipped if you are flushing DITI adapters and Volume in Cleaner is ignored in this case.

- **cleanerDelay** –

- **Airgap** –

- **airgapSpeed** –

- **retractSpeed** –

- **FastWash** –

- **lowVolume** –

- **atFrequency** –

- **RackName** –

- **Well** –

- **arm** –

validate_arg()
> Evoware visual script generator enforce a compatibility between the arguments mask_tip and well selection. If they are not compatible the robot crash. :return:

class instructions.**waste**(*action=0*)
> Bases: EvoScriPy.Instruction_Base.Instruction

> A.15.4.15 Waste (Worklist: Waste)

> **__init__**(*action=0*)
> > Initialize self. See help(type(self)) for accurate signature.

> **actions = range(0, 6)**

> **activate_waste_1 = 1**

> **activate_waste_2 = 2**

> **activate_waste_3 = 3**

> **deactivate_all_wastes = 4**

> **deactivate_system = 5**

> **init_system = 0**

> **validate_arg**()

RobotEvo "modes" for execution of basic instructions

Define how we want to "interact" with the physical robot, or what kind of output we want from

  this script generator.

  - *Mode*
  - *ToString*: an string representation of the instructions.
  - *Multiple*: A collection (list) of all the "modes" to be generated in a single run
  - *ToFile*:
  - *Comments*:
  - *AdvancedWorkList*:
  - *ScriptBody*:
  - *Script*:
  - *iRobot*: update the state of the "internal" robot to track changes produced by the execution of the instructions.

  after each instruction. - *AdvancedWorkList*:

class evo_mode.**AdvancedWorkList**(*filename=None*, *immediate=None*)
  Bases: *evo_mode.ToFile*

  exec(*instr*)

class evo_mode.**COM_automation**
  Bases: *evo_mode.Mode*

class evo_mode.**Comments**(*identation_char=None*, *identattion_length=None*, *current_identation=None*, *filename=None*)
  Bases: *evo_mode.ToFile*

  Create a list with all (and only with) the comments and the Groups. Useful to be shown immediately after generation, but also to the final user just before the actual physical run.

> **__init__**(*identation_char=None*, *identattion_length=None*, *current_identation=None*, *file-*
> *name=None*)
> > Initialize self. See help(type(self)) for accurate signature.

> **exec**(*instr*)

class evo_mode.**Mode**
> Bases: object

> (Base class) Define how we want to "interact" with the physical robot, or what kind of output we want from this script generator. Some options are: A worklist; a full Evoware script; only comments, etc. One important option is to create many of this outputs from a single run.

> **done**()

> **encoding = 'Latin-1'**

> **exec**(*instr*)

> **newline = '\r\n'**

class evo_mode.**Multiple**(*modes=None*)
> Bases: *evo_mode.Mode*

> A collection (list) of all the "modes" to be generated in a single run

> **__init__**(*modes=None*)
> > Initialize self. See help(type(self)) for accurate signature.

> **add_mode**(*mode*)

> **done**()

> **exec**(*instr*)

class evo_mode.**Script**(*filename=None*, *template=None*, *robot_protocol=None*, *robot=None*)
> Bases: *evo_mode.ScriptBody*

> Create a full and executable script for the evoware soft. Take an existing script or script-template as a base.

> **__init__**(*filename=None*, *template=None*, *robot_protocol=None*, *robot=None*)
> > Initialize self. See help(type(self)) for accurate signature.

> **add_template**()

> **exec**(*instr*)

> **set_template**(*template*, *robot_protocol*)

class evo_mode.**ScriptBody**(*filename=None*, *immediate=None*)
> Bases: *evo_mode.ToFile*

class evo_mode.**StdOut**
> Bases: *evo_mode.ToString*

> Specially useful during debugging.

> **exec**(*instr*)

class evo_mode.**ToFile**(*filename=None*, *immediate=None*)
> Bases: *evo_mode.ToString*

> (Base class) For modes with uses a file for output

> **__init__**(*filename=None*, *immediate=None*)
> > Initialize self. See help(type(self)) for accurate signature.

> **done**()

---

exec(*instr*)

open()

set_file(*filename=None*)

class evo_mode.**ToString**

Bases: *evo_mode.Mode*

(Base class) Create an string representation of the instructions.

exec(*instr*)

class evo_mode.**iRobot**(*index*, *n_tips*, *arms=None*, *tips_type=None*)

Bases: *evo_mode.Mode*

Used to validate instructions based on an the state of an internal model af the physical robot. It will check the kind and number of tips, and the volume already aspired in each tips, and the existence and current volume in wells in labware, etc. One basic use of this, is to guarantee that the robot will be actualized once and only once even when multiple modes are used.

__init__(*index*, *n_tips*, *arms=None*, *tips_type=None*)

Initialize self. See help(type(self)) for accurate signature.

exec(*instr*)

set_as_current()

Robots and arms

- *Arm*

- *Robot*: track state to make organizations previous to the actual instruction call, and change that state

after each instruction.

---

class robot.**Arm**(*n_tips*, *index*, *workingTips=None*, *tips_type=None*)

Bases: object

Aspirate = 1

Detect = 0

DiTi = 0

Dispense = -1

Fixed = 1

__init__(*n_tips*, *index*, *workingTips=None*, *tips_type=None*)

Parameters

- **n_tips** – the number of possible tips

- **index** – int. for example: index=Pipette.LiHa1

- **workingTips** – some tips maybe broken or permanently unused.

- **tips_type** – DITI or fixed (not implemented)

eject_tips_executed(*tip_mask=None*) *-> (<class 'int'>, <class 'list'>)*

Drop tips only if needed. Return the mask and the tips really used. :param tip_mask: int :return: the mask that can be used with, is "True" if tips actually ned to be drooped :rtype : int

eject_tips_test(*tip_mask=None*) *-> (<class 'int'>$[$, <class 'int'>$]$)*

Return the mask and the tips index to be really used. :param tip_mask: int :return: the mask that can be used with, is "True" if tips actually ned to be drooped :rtype : int

getMoreTips_test (*rack_type*, *tip_mask=None*) → int
> Mount only the tips with are not already mounted. Mount only one kind of tip at a time, but not necessary the same of the already mounted.
>
> > :rtype : int :param tip_mask: int :return: the mask that can be used

getTips_test (*tip_mask=None*) → int
> Simple test that the asked positions are free for mounting new tips. :rtype : int :param tip_mask: :return: the mask that can be used :raise "Tip already in position " + str(i):

mount_more_tips_executed (*rack_type*, *tip_mask=None*, *tips=None*) -> (*<class 'int'>*, *<class 'list'>*)
> Mount only the tips with are not already mounted. Mount only one kind of tip at a time, but not necessary the same of the already mounted.
>
> > :rtype : int :param tip_mask: int :return: the mask that can be used

mount_tips_executed (*rack_type=None*, *tip_mask=None*, *tips=None*) -> (*<class 'int'>*, *<class 'list'>*)
> Mount only one kind of new tip at a time or just the tips given in the list :param rack_type: :param tips: :rtype : int :param tip_mask: :return: the mask that can be used :raise "Tip already in position " + str(i):

pipette_executed (*action*, *volume*, *tip_mask=None*) -> (*<class 'list'>*, *<class 'int'>*)
> Check and actualize the robot Arm state to aspirate [vol]s with a tip mask. Using the tip mask will check that you are not trying to use an unmounted tip. *volume* values for unsettled tip mask are ignored.
>
> > :rtype : (list, int) :param action: +1:aspirate, -1:dispense :param volume: one vol for all tips, or a list of vol :param tip_mask: -1:all tips :return: a lis of vol to pipette, and the mask

class robot.**Robot** (*index=None*, *arms=None*, *n_tips=None*, *workingTips=None*, *tips_type=0*, *templateFile=None*)
> Bases: object

> Maintain an intern state. Can have more than one arm in a dictionary that map an index with the actual arm. One of the arms can be set as "current" and is returned by cur_arm() Most of the changes in state are made by the implementation of the low level instructions, while the protocols can "observe" the state to make all kind of optimizations and organizations previous to the actual instruction call

> **__init__** (*index=None*, *arms=None*, *n_tips=None*, *workingTips=None*, *tips_type=0*, *templateFile=None*)
> > A Robot may have 1 or more Arms, indexes by key index in a dictionary of Arms. :param arms: :param n_tips: :param workingTips: :param tips_type:

> **cur_arm** (*arm=None*)

> **current = None**

> **drop_tips_executed** (*TIP_MASK=None*, *waste=None*)

> **drop_tips_test** (*TIP_MASK=None*)

> **getTips_test** (*rack_type*, *tip_mask=None*) → int

> **get_tips_executed** (*rack_series*, *tip_mask=None*) -> (*<class 'int'>*, *<class 'list'>*)
> > To be call from instructions.actualize_robot_state(self): actualize iRobot state (tip mounted and DiTi racks) Return the mask with will be really used taking into account the iRobot state, specially, the "reusetips" status and the number of tips already mounted. If it return mask = 0 no evo-instruction for the real robot will be generated in some cases.
> >
> > > **Parameters**
> > > - **tip_mask** –
> > > - **rack_series** – the series of this king of tips.

Returns (int, [labware.Tip])

move_labware_executed(*labware*, *destination*)

pick_up_tips_executed(*TIP_MASK*, *labware_selection: EvoScriPy.labware.DITIrack*) → int
> The low level instruction have to be generated already with almost all the information needed. Here we don't check any more from where we really need to pick the tips and assume they are all in the same rack. Be careful by manual creation of low level instructions: they are safe if they are generated by protocol instructions (drop_tips(), and preserve and usePreserved were previously set). :param labware_selection: :param TIP_MASK:

pipette_executed(*action*, *volume*, *labware_selection*, *tip_mask=None*) -> (*<class 'list'>*, *<class 'int'>*)

preserve_tips(*preserve=True*) → bool

reuse_tips(*reuse=True*) → bool

set_allow_air(*allow_air=0.0*) → float

set_as_current()

set_drop_tips(*drop=True*) → bool
> Drops the tips at THE END of the whole action? like after distribute of the reagent into various target? :param drop: :return: the previous value

set_tips_back_executed(*TIP_MASK*, *labware_selection*)
> The low level instruction have to be generated already with almost all the information needed. Here we don't check any more where we really need to put the tips. Be careful by manual creation of low level instructions: they are safe if they are generated by protocol instructions (drop_tips(), and preserve and usePreserved were previously set). :param TIP_MASK: :param labware:

set_worktable(*templateFile*, *robot_protocol*)

use_preserved_tips(*usePreserved=True*) → bool

use_tips_executed(*tipMask*, *labware_selection*)

where_are_preserved_tips(*selected_reagents: EvoScriPy.labware.Labware*, *TIP_MASK*, *type*) → list

> **Parameters**
>
> > • **selected_reagents** –
> >
> > • **TIP_MASK** –
>
> **Returns** Return a list of racks with the tips-wells already selected.

where_preserve_tips(*TIP_MASK*) → list
> There are used tips in the arm, and we want to know were to put it back. Return a list of racks with the tips-wells already selected. (to set back the tips currently in the arm)

> **Parameters** **TIP_MASK** –

> **Returns** list of racks with the tips-wells already selected.

class instructions_Te_MagS.**Te_MagS_ActivateHeater**(*temperature*, *needs_allwd_lw=0*, *allowed_labware=''*)
> Bases: EvoScriPy.Instruction_Base.TMagInstr

> A.15.10.2 ActivateHeater (Worklist: Te-MagS_ActivateHeater)

> **__init__**(*temperature*, *needs_allwd_lw=0*, *allowed_labware=''*)
> > Initialize self. See help(type(self)) for accurate signature.

> **validate_arg**()

class instructions_Te_MagS.**Te_MagS_DeactivateHeater**(*exec_parameters=",
needs_allwd_lw=0, al-
lowed_labware=")*

Bases: EvoScriPy.Instruction_Base.TMagInstr

A.15.10.3 DeactivateHeater (Worklist: Te-MagS_DeactivateHeater)

__init__(*exec_parameters=", needs_allwd_lw=0, allowed_labware="*)
Initialize self. See help(type(self)) for accurate signature.

validate_arg()

class instructions_Te_MagS.**Te_MagS_Execution**(*exec_parameters=[], needs_allwd_lw=0,
allowed_labware="*)

Bases: EvoScriPy.Instruction_Base.TMagInstr

A.15.10.4 Execution (Worklist: Te-MagS_Execution)

class Parametr(*num*)
Bases: object

__init__(*num*)
Initialize self. See help(type(self)) for accurate signature.

__init__(*exec_parameters=[], needs_allwd_lw=0, allowed_labware="*)
Initialize self. See help(type(self)) for accurate signature.

class command(*firmware_command*)
Bases: object

__init__(*firmware_command*)
Initialize self. See help(type(self)) for accurate signature.

class incub(*hh*, *mm*, *ss*)
Bases: object

__init__(*hh*, *mm*, *ss*)
Initialize self. See help(type(self)) for accurate signature.

class mix(*cycles*, *hh*, *mm*, *ss*, *z_pos=31*)
Bases: object

__init__(*cycles*, *hh*, *mm*, *ss*, *z_pos=31*)
Initialize self. See help(type(self)) for accurate signature.

class move(*position*, *z_pos*)
Bases: object

__init__(*position*, *z_pos*)
Initialize self. See help(type(self)) for accurate signature.

validate_arg()

class wait(*hh*, *mm*, *ss*)
Bases: object

__init__(*hh*, *mm*, *ss*)
Initialize self. See help(type(self)) for accurate signature.

class instructions_Te_MagS.**Te_MagS_MoveToPosition**(*position*, *z_pos=31*,
needs_allwd_lw=0, *al-
lowed_labware="*)

Bases: EvoScriPy.Instruction_Base.TMagInstr

A.15.10.1 MoveToPosition (Worklist: Te-MagS_MoveToPosition)

```
Aspirate = 1
```

```
Dispense = 0
```

```
Incubation = 3
```

```
Re_suspension = 2
```

__init__ (*position*, *z_pos=31*, *needs_allwd_lw=0*, *allowed_labware=''*)

> **Parameters**
>
> > • **position** – Aspirate Position - Carrier above the magnet block, magnet block raised. Dispense Position - Carrier above the magnet block, magnet block lowered. Incubation Position - Carrier above the heating block, heating block raised. Re-suspension Position - Carrier above the heating block, heating block lowered.
> >
> > > Use this position to carry out re-suspension by mixing the liquid with the pipetting tips (e.g. with the LiHa - Mix script command).
> >
> > • **z_pos** –
> >
> > • **needs_allwd_lw** –
> >
> > • **allowed_labware** –

validate_arg()

CHAPTER 12

Examples:

Indices and tables

- genindex

- modindex

- search

Python Module Index

e
evo_mode, 73

i
instructions, 55
instructions_Te_MagS, 81

l
labware, 45

p
protocol_steps, 21

r
reagent, 32
robot, 77

Index

Symbols

O

P

Q

R

S

W

X

Y

Z