AFL-automation

NIST AFL Team

May 08, 2025

GETTING STARTED

AFL-automation is a framework for instrument control and laboratory automation. It powers the NIST AFL (Autonomous Formulation Laboratory), but is designed to be versatile for many scientific instrumentation needs. It enables the easy conversion of Python classes - drivers - into robust HTTP microservices with authentication, task queueing, UI generation, data management, and more.

CHAPTER

ONE

INSTALLING AFL-AUTOMATION

This tutorial will guide you through the process of installing AFL-automation and its dependencies.

1.1 Basic Installation

AFL-automation can be installed using pip:

```
pip install AFL-automation
```

This will install the core dependencies needed for basic functionality.

1.2 Installation with Hardware Support

Depending on your specific hardware needs, you may want to install additional dependencies:

```
# For Ocean Insight spectrometers
pip install AFL-automation[seabreeze]

# For Opentrons liquid handling robots
pip install AFL-automation[opentrons]

# For multiple hardware types
pip install AFL-automation[seabreeze,opentrons]
```

For a complete list of available extras and what they provide, see the *Managing Dependencies* page.

1.3 Development Installation

For development, you might want to install in editable mode with additional tools:

```
git clone https://github.com/usnistgov/AFL-automation.git
cd AFL-automation
pip install -e .

# Install development tools
pip install -e ".[docs]"
```

CHAPTER

TWO

QUICK START GUIDE

This quick start guide will help you get up and running with AFL-automation quickly.

2.1 Creating Your First Driver

AFL-automation is built around the concept of *drivers* - Python classes that interact with hardware or provide services. Here's a simple example:

```
from AFL.automation.APIServer.Driver import Driver

class SimpleDriver(Driver):
    defaults = {}
    defaults['greeting'] = 'Hello, World!'

    def __init__(self, overrides=None):
        Driver.__init__(self, name='SimpleDriver',
                        defaults=self.gather_defaults(),
                        overrides=overrides)

    def say_hello(self):
        """Say a greeting based on configuration"""
        return self.config['greeting']
```

2.2 Running as a Microservice

Once you have a driver, you can turn it into a web service:

```
from AFL.automation.APIServer.APIServer import APIServer

# Create the driver instance
my_driver = SimpleDriver()

# Create the server
server = APIServer('my-service')

# Add your driver and run the server
server.create_queue(my_driver)
server.run(host='0.0.0.0', port=5000)
```

2.3 Accessing the Service

You can now access your service via HTTP requests or use the built-in client:

```
from AFL.automation.APIServer.Client import Client

# Connect to the service
client = Client('http://localhost:5000')

# Call a method
response = client.enqueue(task_name='say_hello',interactive=True)
print(response)  # Outputs: 'Hello, World!'

# Call a method asynchronously
response = client.enqueue(task_name='say_hello',interactive=False)
print(response)  # Outputs a uuid

# Get the result of the task
result = client.get_result(response)
print(result)  # Outputs: 'Hello, World!'
```

2.4 Next Steps

For more detailed instructions, check out:

- *AFL-automation Architecture* - Learn about AFL-automation's architecture
- *Managing Dependencies* - Manage hardware-specific dependencies
- API Reference - Explore the full API documentation

CHAPTER

THREE

TUTORIALS

Tutorials are learning-oriented guides that help new users get started with AFL-automation.

CHAPTER

FOUR

HOW-TO GUIDES

How-to guides are problem-oriented instructions that help users accomplish specific tasks with AFL-automation.

4.1 Managing Dependencies

The AFL-automation package uses a modular dependency system to handle various hardware and specialized libraries. This allows you to install only the dependencies you need for your specific hardware configuration.

4.1.1 Core Dependencies

The core dependencies are installed automatically when you install the package. These include:

- **Web framework components**: Flask, Flask-CORS, Flask-JWT-Extended
- **Data processing libraries**: NumPy, pandas, SciPy, xarray, h5py
- **Visualization tools**: Matplotlib, Plotly, Bokeh, ipywidgets
- **Scientific utilities**: periodictable, scikit-learn, scikit-image

4.1.2 Optional Dependencies (Extras)

The package uses Python's "extras" mechanism to manage optional dependencies. You can install these using pip:

```
# Install with specific hardware support
pip install AFL-automation[seabreeze,opentrons]

# Install with scattering support
pip install AFL-automation[scattering-processing,sas-analysis]
```

4.1.3 Available Extras

Hardware Interfaces

Extra Name	Package(s)	Description
`labjack`	labjack-ljm	Support for LabJack data acquisition hardware
`piplates`	piplates	Support for Pi-Plates hardware add-ons
`rpi-gpio`	RPi.GPIO	Raspberry Pi GPIO interface libraries
`serial`	pyserial	Serial communication support for instruments
`seabreeze`	seabreeze	Support for Ocean Optics/Ocean Insight spectrometers
`opentrons`	opentrons	Support for Opentrons liquid handling robots
`pyspec`	certif-pyspec	Support for CHESS and other beamline control
`remote-access`	paramiko	SSH client for remote instrument connections

Scientific Processing

Extra Name	Package(s)	Description
`scattering-proces`	fabio, pyFAI	Scattering data processing and azimuthal integration
`sas-analysis`	sasmodels, sasdata	Small-Angle Scattering analysis tools
`ml`	tensorflow, gpflow	Machine learning utilities for data analysis
`geometry`	alphashape, shapely	Geometric analysis and manipulation tools

Neutron Scattering

Extra Name	Package(s)	Description
`neutron-scatterin`	epics, sans, mantid	Support for neutron scattering instrumentation
`nice-neutron-scat`	nice	NIST NCNR NICE control system

Documentation

Extra Name	Package(s)	Description
`docs`	sphinx, sphinx-rtd-theme, sphinx-autodoc-typehints, sphinx-copybutton, myst-parser, nbsphinx	Tools for building documentation

4.1.4 Implementation Details

AFL-automation uses lazy loading for optional dependencies. This means that:

1. You can import the package without having all optional dependencies installed
2. Dependencies are only loaded when actually used (via the `lazy_loader` library)
3. Clear error messages will tell you which extra to install if you try to use a feature without its required dependency

Example of error when trying to use a feature without the required dependency:

```
>>> from AFL.automation.instrument import SeabreezeUVVis
>>> spectrometer = SeabreezeUVVis()
ImportError: This module requires the 'seabreeze' package.
Please install it: pip install AFL-automation[seabreeze]
```

4.1.5 Importing Multiple Hardware Packages

If you need support for multiple hardware types, you can specify multiple extras:

```
pip install AFL-automation[seabreeze,opentrons,scattering-processing]
```

Core packages that are not tied to specific hardware will be loaded normally without any special handling.

CHAPTER

FIVE

EXPLANATION

Explanation guides are understanding-oriented articles that provide background context and explain key concepts of AFL-automation.

5.1 AFL-automation Architecture

This page explains the core architecture of AFL-automation and how its components work together.

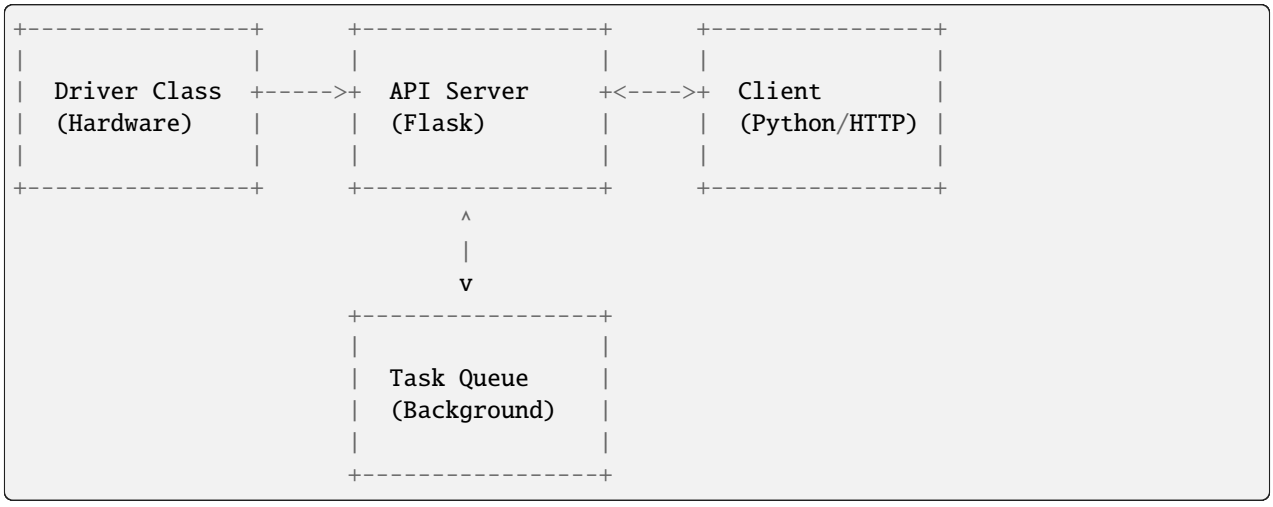
5.1.1 Core Concepts

AFL-automation is built around several key concepts:

1. **Drivers**: Python classes that interface with hardware or provide services
2. **APIServer**: A Flask-based web server that exposes drivers via HTTP
3. **Task Queue**: A system for managing asynchronous tasks
4. **Client**: A Python client for interacting with remote services

5.1.2 Architecture Diagram

The following diagram illustrates the high-level architecture:



5.1.3 Driver System

The driver system is the core of AFL-automation. Drivers:

- Encapsulate hardware control logic
- Define configuration parameters with defaults
- Provide methods for interacting with hardware
- Support lazy loading of hardware-specific dependencies

5.1.4 API Server

The API server:

- Exposes driver methods via HTTP endpoints
- Provides authentication and authorization
- Manages a task queue for asynchronous operations
- Offers a web UI for monitoring and control

5.1.5 Dependency Management

AFL-automation uses a modular dependency system:

- Core dependencies are always installed
- Hardware-specific dependencies are optional
- Lazy loading ensures code works even without all dependencies
- The extras system makes installation straightforward

For details on managing dependencies, see *Managing Dependencies*.

CHAPTER

SIX

REFERENCE

Reference documentation provides information-oriented technical descriptions of the AFL-automation API and code structure.

6.1 API Reference

This page provides detailed API documentation for all modules in the AFL-automation framework.

6.1.1 AFL.automation

Functions

test()	Run all tests using pytest.

AFL.automation.**test**()

Run all tests using pytest.

Modules

APIServer
EpicsADLiveProcess
instrument
loading
prepare
sample_env
shared

AFL.automation.APIServer

Modules

APIServer
Client
Driver
DummyDriver
DummyOT2Driver
LoggerFilter
QueueDaemon

AFL.automation.APIServer.APIServer

Functions

create_access_token(identity[, fresh, ...])	Create a new access token.
create_refresh_token(identity[, ...])	Create a new refresh token.
get_jwt_identity()	In a protected endpoint, this will return the identity of the JWT that is accessing the endpoint.
jsonify(*args, **kwargs)	Serialize the given arguments as JSON, and return a `Response` object with the `application/json` mimetype.
jwt_required([optional, fresh, refresh, ...])	A decorator to protect a Flask endpoint with JSON Web Tokens.
listify(obj)	
render_template(template_name_or_list, **context)	Render a template by name with the given context.
send_file(path_or_file[, mimetype, ...])	Send the contents of a file to the client.
set_access_cookies(response, ...[, max_age, ...])	Modifiy a Flask Response to set a cookie containing the access JWT.
set_refresh_cookies(response, ...[, ...])	Modifiy a Flask Response to set a cookie containing the refresh JWT.
strtobool(val)	Convert a string representation of truth to true (1) or false (0).
unset_jwt_cookies(response[, domain])	Modifiy a Flask Response to delete the cookies containing access or refresh JWTs.

Classes

APIServer(name[, data, experiment, contact, ...])	
CORS([app])	Initializes Cross Origin Resource sharing for the application.
FileHandler(filename[, mode, encoding, ...])	A handler class which writes formatted logging records to disk files.
Flask(import_name[, static_url_path, ...])	The flask object implements a WSGI application and acts as the central object.
IPVersion(value[, names, module, qualname, ...])	
JWTManager([app, add_context_processor])	An object used to hold JWT settings and callback functions for the Flask-JWT-Extended extension.
LoggerFilter(*filters)	
MutableQueue()	Thread-safe, mutable queue
QueueDaemon(app, driver, task_queue, history)	
SMTPHandler(mailhost, fromaddr, toaddrs, subject)	A handler class which sends an SMTP email for each logging event.
ServiceInfo	Service information.
Zeroconf([interfaces, unicast, ip_version, ...])	Implementation of Zeroconf Multicast DNS Service Discovery

class AFL.automation.APIServer.APIServer.**APIServer**(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

__init__(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

create_queue(*driver*, *add_unqueued=True*)

reset_queue_daemon(*driver=None*)

advertise_zeroconf(***kwargs*)

run(***kwargs*)

run_threaded(*start_thread=True*, ***kwargs*)

add_standard_routes()

get_info()

Live, status page of the robot

get_quickbar()

Return the functions, params, and defaults to be shown in this server's quickbar

is_server_live()

get_unqueued_commands()

get_queued_commands()

add_unqueued_routes()

query_driver()

init_logging(*toaddrs=None*)

index()

Live, status page of the robot

index_new()

Live, status page of the robot

webapp()

Live, status page of the robot

render_unqueued(*func*, *kwargs_add*, ***kwargs*)

Convert an unqueued return item into web-suitable output

send_1d_plot(*result*, *multi=False*, ***kwargs*)

send_array_as_jpg(*array*, *log_image=False*, *max_val=None*, *fillna=0.0*, ***kwargs*)

queue_state()

driver_status()

get_queue()

get_queue_iteration()

deposit_obj()

Store an object named obj in the driver's dropbox If a uuid is provided, the object will be stored with that uuid Otherwise, a new uuid will be generated. In either case, the uuid will be returned to the client.

retrieve_obj()

Retrieve an object from the driver's dropbox uuid specifies the object to retrieve delete specifies whether to delete the object after retrieval

set_driver_object()

get_driver_object()

enqueue()

reorder_queue()

remove_items()

remove_item()

move_item()

clear_queue()

clear_history()

debug()

pause()

halt()

init()

login()

get_server_time()

login_test()

AFL.automation.APIServer.Client

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
ServerDiscovery()	ServerDiscovery class

class AFL.automation.APIServer.Client.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.APIServer.Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
makeRegistrar()	
sqrt(x, /)	Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
PersistentConfig(path[, defaults, ...])	A dictionary-like class that serializes changes to disk

AFL.automation.APIServer.Driver.**makeRegistrar**()

class AFL.automation.APIServer.Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.APIServer.DummyDriver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

Classes

Driver(name[, defaults, overrides])
DummyDriver([name, overrides])

class AFL.automation.APIServer.DummyDriver.**DummyDriver**(*name=None*, *overrides=None*)

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

test_command1(*kwarg1=None*, *kwarg2=True*)

A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

AFL.automation.APIServer.DummyOT2Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

Classes

Driver(name[, defaults, overrides])
DummyDriver([name, overrides])

class AFL.automation.APIServer.DummyOT2Driver.**DummyDriver**(*name=None*, *overrides=None*)

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

execute(***kwargs*)

get_prep_target(***kwargs*)

test_command1(*kwarg1=None*, *kwarg2=True*)

A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

AFL.automation.APIServer.LoggerFilter

Classes

LoggerFilter(*filters)	

class AFL.automation.APIServer.LoggerFilter.**LoggerFilter**(**filters*)

__init__(**filters*)

AFL.automation.APIServer.QueueDaemon

Functions

is_serialized(obj)	

Classes

DataTrashcan()	A DataPacket implementation *for testing only* that takes all its data and simply throws it away.
QueueDaemon(app, driver, task_queue, history)	

class AFL.automation.APIServer.QueueDaemon.**QueueDaemon**(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

check_if_paused()

mask_serialized_objs(*package*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.EpicsADLiveProcess

Modules

AreaDetectorLive
Client
ReduceDaemon

AFL.automation.EpicsADLiveProcess.AreaDetectorLive

Classes

AreaDetectorLive([basepv, cam, filewriter, ...])

class AFL.automation.EpicsADLiveProcess.AreaDetectorLive.**AreaDetectorLive**(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

__init__(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

cbfunc(*pvname=None*, *value=None*, *char_value=None*, ***kwargs*)

queuehandler()

status()

AFL.automation.EpicsADLiveProcess.Client

Classes

Client([ip, port])	Communicate with NistoRoboto server on OT-2

class AFL.automation.EpicsADLiveProcess.Client.**Client**(*ip='10.42.0.30'*, *port='5000'*)

Communicate with NistoRoboto server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the NistoRoboto server

__init__(*ip='10.42.0.30'*, *port='5000'*)

logged_in()

login(*username*)

set_queue_mode(*debug_mode=True*)

transfer(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str*) – Source wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

reset()

home()

halt()

AFL.automation.EpicsADLiveProcess.ReduceDaemon

Classes

ReduceDaemon(app, reduction_queue, ...[, ...])	

class AFL.automation.EpicsADLiveProcess.ReduceDaemon.**ReduceDaemon**(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

__init__(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.instrument

Modules

DummySAS
FileCamera
I22SAXS
NetworkCamera
SeabreezeUVVis
SpecScreen_Driver

AFL.automation.instrument.DummySAS

Classes

Driver(name[, defaults, overrides])
DummySAS([overrides])

class AFL.automation.instrument.DummySAS.**DummySAS**(*overrides=None*)

defaults = {}

__init__(*overrides=None*)

connect to spec

expose(*name=None*, *exposure=None*, *nexp=1*, *block=True*, *reduce_data=True*, *measure_transmission=True*, *save_nexus=True*)

status()

AFL.automation.instrument.FileCamera

Classes

FileCamera()	

class AFL.automation.instrument.FileCamera.**FileCamera**

__init__()

collect(*fname*)

AFL.automation.instrument.I22SAXS

Classes

Driver(name[, defaults, overrides])	
I22SAXS([overrides])	

class AFL.automation.instrument.I22SAXS.**I22SAXS**(*overrides=None*, ***kwargs*)

defaults = {'acq_time': 1, 'acq_timeout': 120, 'address': '', 'data_read_cooldown': 5, 'empty_scan_id': '', 'file_read_timeout': 30, 'nframes': 1, 'port': 2222, 'pos_list': [], 'processed_base_path': '/mnt/i22_processed/i22-', 'reduced_data_suffix': '_saxs_Transmission_Averaged_Subtracted_IvsQ_processed.nxs', 'ssh_key_path': '', 'username': ''}

__init__(*overrides=None*, ***kwargs*)

expose(*name*, *empty=False*, *nframes=None*, *acq_time=None*, *set_empty=False*)

Perform a sequence of exposures at positions defined in self.config['pos_list'].

Return the integrated data of the measurement with the lowest measured sample transmission.

read_integrated(*scanid*)

Scans the appropriate directory for the filename of the data collected with filename

AFL.automation.instrument.NetworkCamera

Classes

NetworkCamera(url)	

class AFL.automation.instrument.NetworkCamera.**NetworkCamera**(*url*)

__init__(*url*)

camera_reset()

collect()

AFL.automation.instrument.SeabreezeUVVis

Classes

Driver(name[, defaults, overrides])	
Eq(key, value)	Query equality of a given key's value to the specified value.
Path(*args, **kwargs)	PurePath subclass that can make system calls.
SeabreezeUVVis([backend, device_serial, ...])	

class AFL.automation.instrument.SeabreezeUVVis.**SeabreezeUVVis**(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

defaults = {'air_uuid': '', 'correctDarkCounts': False, 'correctNonlinearity': False, 'exposure': 0.01, 'exposure_delay': 0, 'filename': 'test.h5', 'filepath': '.', 'reference_uuid': '', 'saveSingleScan': False}

__init__(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

getExposure()

getExposureDelay()

getFilename()

getSaveSingleScan()

getFilepath()

setFilepath(*filepath*)

setFilename(*filename*)

setSaveSingleScan(*saveSingleScan*)

setExposureDelay(*time*)

setExposure(*time*)

collectContinuous(*duration*, *start=None*, *return_data=False*, ***kwargs*)

collectSingleSpectrum(*set_reference=False*, *set_air=False*, ***kwargs*)

collect(*nframes: int*, *reduced: bool = False*, *absorbance: bool = True*, *set_reference: bool = False*, *set_air: bool = False*, *exposure: float | None = None*, *return_data: bool = False*, ***kwargs*)

reduced(*data_raw_mean*, *data_raw_std*, *absorbance=True*, *reference_uuid='reference_uuid'*)

AFL.automation.instrument.SpecScreen_Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
SpecScreen_Driver([log_file])	

class AFL.automation.instrument.SpecScreen_Driver.**SpecScreen_Driver**(*log_file=None*)

__init__(*log_file=None*)

status()

execute(***kwargs*)

AFL.automation.loading

Modules

CetoniMultiPosValve	
ChemyxSyringePump	This is largely duplicated from the reference code provided by Chemyx.
DigitalOutPressureController	
DoubleViciMultiposSelector	
DummyPump	
FlowSelector	
LoadStopperDriver	
MultiChannelRelay	
NE1kSyringePump	
OneSelectorBlowoutSampleCell	
PneumaticPressureSampleCell	
PneumaticSampleCell	
PressureController	
PressureControllerAsPump	
PushPullSelectorSampleCell	
RSoXSSolutionSampleCell	
SainSmartRelay	
SampleCell	
Sensor	
SensorCallbackThread	
SensorPollingThread	
SerialDevice	
SyringePump	
Tubing	
TwoSelectorBlowoutSampleCell	
UltimusVPressureController	
ViciMultiposSelector	

AFL.automation.loading.CetoniMultiPosValve

Classes

CetoniMultiPosValve(parentpump[, portlabels])	
FlowSelector()	

class AFL.automation.loading.CetoniMultiPosValve.**CetoniMultiPosValve**(*parentpump*, *portlabels={}*)

__init__(*parentpump*, *portlabels={}*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.ChemyxSyringePump

This is largely duplicated from the reference code provided by Chemyx. Their package is a GUI and direct import of the module would be problematic.

Functions

getOpenPorts()	
parsePortName(portinfo)	On macOS and Linux, selects only usbserial options and parses the 8 character serial number.

Classes

ChemyxConnection(port, baudrate[, x, mode, ...])	
ChemyxSyringePump(port, syringe_id_mm, ...)	
SyringePump()	

class AFL.automation.loading.ChemyxSyringePump.**ChemyxSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

Initializes and verifies connection to a Chemyx syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

wait_dosage_finished(*timeout_seconds=60*)

The function waits until the last dosage command has finished until the timeout occurs.

stop()

Abort the current dispense/withdraw action.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

getLevel()

setLevel(*level*)

blockUntilStatusStopped(*pollingdelay=0.2*)

This is a deprecated function from old serial logic. It should work, but do not use.

getStatus()

query the pump status and return whether the pump is moving or not (true if moving, false if stopped)

getValueFromParams(*search_key*)

AFL.automation.loading.ChemyxSyringePump.**getOpenPorts**()

AFL.automation.loading.ChemyxSyringePump.**parsePortName**(*portinfo*)

On macOS and Linux, selects only usbserial options and parses the 8 character serial number.

class AFL.automation.loading.ChemyxSyringePump.**ChemyxConnection**(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

__init__(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

openConnection()

closeConnection()

sendCommand(*command*)

getResponse()

startPump()

stopPump()

pausePump()

restartPump()

setUnits(*units*)

setDiameter(*diameter*)

setRate(*rate*)

setVolume(*volume*)

setDelay(*delay*)

setTime(*timer*)

getParameterLimits()

getParameters()

getDisplacedVolume()

getElapsedTime()

getPumpStatus()

addMode(*command*)

addX(*command*)

AFL.automation.loading.DigitalOutPressureController

Classes

DigitalOutPressureController(digital_out, ...)	
PressureController()	Abstract superclass for pressure controllers that provides timed dispensing

class AFL.automation.loading.DigitalOutPressureController.**DigitalOutPressureController**(*digital_out*, *pressure_to_v_conv*)

__init__(*digital_out*, *pressure_to_v_conv*)

Initializes a DigitalOutPressureController

Params:

digital_out (AFL.automation.DigitalOut): pressure_to_v_conv (float): pressure units per volt

set_P(*pressure*)

AFL.automation.loading.DoubleViciMultiposSelector

Classes

DoubleViciMultiposSelector(port1, port2[, ...])	
FlowSelector()	
ViciMultiposSelector(port[, baudrate, ...])	

class AFL.automation.loading.DoubleViciMultiposSelector.**DoubleViciMultiposSelector**(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

__init__(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.DummyPump

Classes

DummyPump()	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

class AFL.automation.loading.DummyPump.**DummyPump**

__init__()

Dummy pump for testing - does nothing, but boy does it look good doing it.

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.FlowSelector

Classes

FlowSelector()	

class AFL.automation.loading.FlowSelector.**FlowSelector**

getPort()

selectPort()

AFL.automation.loading.LoadStopperDriver

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
Driver(name[, defaults, overrides])	
LoadStopperDriver(sensor[, load_client, ...])	Driver for stopping loads
SensorPollingThread(sensor[, period, ...])	
StopLoadCBv1(poll, period, load_client[, ...])	
StopLoadCBv2(poll, period[, load_client, ...])	

class AFL.automation.loading.LoadStopperDriver.**LoadStopperDriver**(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

Driver for stopping loads

defaults = {'load_speed': 2, 'period': 0.05, 'poll_window': 1000, 'sensorlabel': '', 'stopper_baseline_duration': 2, 'stopper_filepath': '/github/home/.afl/loadstopper_data', 'stopper_loadstop_cooldown': 2, 'stopper_min_load_time': 3, 'stopper_post_detection_sleep': 1, 'stopper_threshold_npts': 50, 'stopper_threshold_std': 3.0, 'stopper_threshold_v_step': 1, 'stopper_timeout': 120}

__init__(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

status()

property app

calibrate_sensor()

read_sensor()

read_poll()

read_poll_load()

reset()

reset_poll()

reset_stopper()

AFL.automation.loading.MultiChannelRelay

Classes

MultiChannelRelay()	

class AFL.automation.loading.MultiChannelRelay.**MultiChannelRelay**

setChannels(*channels*)

getChannels(*channels*)

toggleChannels(*channels*)

AFL.automation.loading.NE1kSyringePump

Classes

NE1kSyringePump(port, syringe_id_mm, ...[, ...])	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

class AFL.automation.loading.NE1kSyringePump.**NE1kSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

Initializes and verifies connection to a New Era 1000 syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

daisy_chain = used for the 'party-line' mode on these pumps where a string of pumps is on one serial port.

when setting up daisy chaining:
connect to the first pump on a port with daisy_chain = False on subsequent pumps, set daisy_chain to the pump with a hardware connection (the first pump)

or any other pump on the string.

note: when daisy chaining you should probably set pumpid explicitly rather than autodiscovering
as most likely the autodiscovery will return the first pump id each time.

pumpid = the ID configured in the pump firmware. If not set, will attempt to auto-discover a pump.
setting pumpid will save some time on connection and probably result in more reproducible behavior. Practically mandatory for daisy chain mode.

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.OneSelectorBlowoutSampleCell

Classes

Driver(name[, defaults, overrides])	
OneSelectorBlowoutSampleCell(pump, selector)	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
TwoSelectorBlowoutSampleCell(pump, selector, ...)	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**OneSelectorBlowoutSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

AFL.automation.loading.PneumaticPressureSampleCell

Classes

Driver(name[, defaults, overrides])	
PneumaticPressureSampleCell(pctrl, relayboard)	Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.
SampleCell()	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.PneumaticPressureSampleCell.**PneumaticPressureSampleCell**(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=None*, *overrides=None*)

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a variable-pressure regulator.

```
defaults = {'arm_move_delay': 0.2, 'blowout_pressure': 20,
'external_load_complete_trigger': False, 'load_mode': 'static', 'load_pressure':
2, 'load_timeout': 60, 'ramp_load_duration': 20, 'ramp_load_stop_pressure': 7,
'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None,
0.5), ('blow', 5)], 'vent_delay': 0.5}
```

__init__(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=None*, *overrides=None*)

pctrl: a pressurecontroller object supporting the set_P method() and the optional p_ramp() method.
e.g. pctrl = DigitalOutPressureController(LabJackDigitalOut(...))

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

property data

status()

loadSample(*cellname='cell'*, *sampleVolume=None*, *load_dest_label=''*)

Load a sample into the cell

Params *cellname* and *sampleVolume* are kept for backward compat, but are deprecated and unused.

advanceSample(*load_dest_label=''*)

Move a sample from one measurement cell to the next

Params:

load_dest_label (str, default ''): a 'destination label' for this load.
labeled sensors will only stop the load if their name is in this destination label. example:

sensor 1 labeled 'afterSANS' sensor 2 labeled 'beforeSPEC' sensor 3 labeled '' (default)

advanceSample(load_dest_label='afterSANS') –> sensor 1 or sensor 3 can stop it advanceSample(load_dest_label='beforeSPEC afterSANS') –> sensor 1, sensor 2, or sensor 3 can stop it advanceSample(load_dest_label='') –> only sensor 3 can stop it

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

read_sensor()

read_sensor_poll(***kwargs*)

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset(*sensor_n=None*)

AFL.automation.loading.PneumaticSampleCell

Classes

Driver(name[, defaults, overrides])	
PneumaticSampleCell(pump, relayboard[, ...])	Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.
SampleCell()	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.PneumaticSampleCell.**PneumaticSampleCell**(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a syringe pump.

defaults = {'air_speed': 30, 'arm_move_delay': 0.2, 'catch_to_cell_vol': 1.15, 'external_load_complete_trigger': False, 'large_dispense_vol': 5, 'load_speed': 2, 'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None, 0.5), ('blow', 5)], 'vent_delay': 0.5, 'withdraw_vol': 1.5}

__init__(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_pump(*dispense=False*)

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

status()

loadSample(*cellname='cell'*, *sampleVolume=0*)

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

read_sensor()

read_sensor_poll(***kwargs*)

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset()

AFL.automation.loading.PressureController

Classes

PressureController()	Abstract superclass for pressure controllers that provides timed dispensing

class AFL.automation.loading.PressureController.**PressureController**

Abstract superclass for pressure controllers that provides timed dispensing

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

AFL.automation.loading.PressureControllerAsPump

Classes

PressureControllerAsPump(pressure_controller)	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

class AFL.automation.loading.PressureControllerAsPump.**PressureControllerAsPump**(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

__init__(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

Initialize a pressure controller as a syringe pump.

pressure_controller: controller to connect to dispense_pressure: pressure at which dispense should run implied_flow_rate: flow rate which should be used to convert to dispense times, mL/min

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.PushPullSelectorSampleCell

Classes

Driver(name[, defaults, overrides])	
PushPullSelectorSampleCell(pump, selector[, ...])	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
SampleCell()	
Tubing(specid, length)	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.PushPullSelectorSampleCell.**PushPullSelectorSampleCell**(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_v*, *calibrated_syringe_to_cell_vo*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector) @TODO: figure out when/where to pull in air to the syringe to make up for extra_vol_to_empty_ml

__init__(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_vol=None*, *calibrated_syringe_to_cell_vol=None*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

property app

status()

setParameter(*parameter*, *value*)

getParameters()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCell(*cellname='cell'*)

blowOutCellForcedAir(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

AFL.automation.loading.RSoXSSolutionSampleCell

Classes

Driver(name[, defaults, overrides])	
RSoXSSolutionSampleCell(pump, selector[, ...])	Class for a sample cell consisting of a pump and a one-to-many flow selector for a flowrate-limited measurement cell (e.g., a liquid TEM sample holder for RSoXS).
SampleCell()	
Tubing(specid, length)	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.RSoXSSolutionSampleCell.**RSoXSSolutionSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector for a flowrate-limited measurement cell (e.g., a liquid TEM sample holder for RSoXS).

The pump draws from any of the sample ports and can quickly purge/rinse itself and the tubing between the loader and the cell. When the pump is connected to the cell, its max rate is limited to very slow (5 ul/min for RSoXS).

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

setWasteLevel(*vol*)

AFL.automation.loading.SainSmartRelay

Module Attributes

usbrelay

Classes

MultiChannelRelay()
SainSmartRelay(relaylabels, serial_port[, ...])
SerialDevice(port[, baudrate, timeout, ...])

class AFL.automation.loading.SainSmartRelay.**SainSmartRelay**(*relaylabels*, *serial_port*, *timeout=0.5*)

__init__(*relaylabels*, *serial_port*, *timeout=0.5*)

Init connection to a SainSmart 16-channel USB relay module.

Params: relaylabels (dict):

mapping of port id to load name, e.g. {0:'arm_up',1:'arm_down'}

serial_port (str):
port to connect to

setAllChannelsOff()

setChannels(*channels*)

Write a value (True, False) to the channels specified in channels

Parameters: channels (dict):

dict of channels, keys as either str (name) or int (id) to write to, vals as the value to write

getChannels(*asid=False*)

Read the current state of all channels

Parameters: asid (bool,default false): Dict keys should simply be the id, not the name.

Returns: (dict) key:value mappings of state.

toggleChannels(*channels*)

AFL.automation.loading.SainSmartRelay.**usbrelay = [[b':FE0100200000FF\r\n', b':FE0100000010F1\r\n'], [b':FE0500000000FD\r\n', b':FE050000FF00FE\r\n'], [b':FE0500010000FC\r\n', b':FE050001FF00FD\r\n'], [b':FE0500020000FB\r\n', b':FE050002FF00FC\r\n'], [b':FE0500030000FA\r\n', b':FE050003FF00FB\r\n'], [b':FE0500040000F9\r\n', b':FE050004FF00FA\r\n'], [b':FE0500050000F8\r\n', b':FE050005FF00F9\r\n'], [b':FE0500060000F7\r\n', b':FE050006FF00F8\r\n'], [b':FE0500070000F6\r\n', b':FE050007FF00F7\r\n'], [b':FE0500080000F5\r\n', b':FE050008FF00F6\r\n'], [b':FE0500090000F4\r\n', b':FE050009FF00F5\r\n'], [b':FE05000A0000F3\r\n', b':FE05000AFF00F4\r\n'], [b':FE05000B0000F2\r\n', b':FE05000BFF00F3\r\n'], [b':FE05000C0000F1\r\n', b':FE05000CFF00F2\r\n'], [b':FE05000D0000F0\r\n', b':FE05000DFF00F1\r\n'], [b':FE05000E0000FF\r\n', b':FE05000EFF00F0\r\n'], [b':FE05000F0000FE\r\n', b':FE05000FFF00FF\r\n'], [b':FE0F00000010020000E1\r\n', b':FE0F0000001002FFFFE3\r\n']]**

“”” Sainsmart 16-Channel 9-36v USB Relay Module (CH341 chip)(sku# 101-70-208) 1. Requires CH341 Windows driver installed (see http://wiki.sainsmart.com/index.php/101-70-208) 2. The hex table provided by Sainsmart requires converting to ASCII chars (see ‘usbrelay’ below) 3. Format of ‘usbrelay’ two-dimensional array is:

usbrelay = [row][ch-off, ch-on] so that the 2nd index selects ON/OFF value
while the 1st index selects the array row.

Example…
status = usbrelay[0][1] # row-0 (status) stat_ret = usbrelay[0][0] # row-0 (status return) ch_1_on = usbrelay[1][1] # row-1 (chan-1 off) ch_1_off = usbrelay[1][0] # row-1 (chan-1 off) ch_16_on = usbrelay[16][1] # row-16 (chan-16 on) ch_16_off = usbrelay[16][0] # row-16 (chan-16 off) all_on = usbrelay[17][1] # row-17 (all on) all_off = usbrelay[17][0] # row-17 (all off)

“””

AFL.automation.loading.SampleCell

Classes

SampleCell()	

class AFL.automation.loading.SampleCell.**SampleCell**

loadSample()

AFL.automation.loading.Sensor

Classes

DummySensor1([period, minval, maxval])	
DummySensor2([period, hi_value, lo_time, ...])	
Sensor([address, channel])	

class AFL.automation.loading.Sensor.**Sensor**(*address=1*, *channel=0*)

__init__(*address=1*, *channel=0*)

calibrate()

read()

class AFL.automation.loading.Sensor.**DummySensor1**(*period=2*, *minval=-5*, *maxval=5*)

__init__(*period=2*, *minval=-5*, *maxval=5*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.loading.Sensor.**DummySensor2**(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

__init__(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

driver_status()

read()

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.loading.SensorCallbackThread

Classes

LoaderCommunication([load_client, load_object])	
SensorCallbackThread(poll[, period, daemon, ...])	
SimpleThresholdCB(poll, period[, window, ...])	
StopLoadCBv1(poll, period, load_client[, ...])	
StopLoadCBv2(poll, period[, load_client, ...])	

class AFL.automation.loading.SensorCallbackThread.**SensorCallbackThread**(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

update_status(*value*)

terminate()

process_signal(*signal*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv1**(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv2**(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**SimpleThresholdCB**(*poll*, *period*, *window=5*, *threshold=1*)

__init__(*poll*, *period*, *window=5*, *threshold=1*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**LoaderCommunication**(*load_client=None*, *load_object=None*)

__init__(*load_client=None*, *load_object=None*)

getServerState()

stopLoad()

AFL.automation.loading.SensorPollingThread

Classes

SensorPollingThread(sensor[, period, ...])	

class AFL.automation.loading.SensorPollingThread.**SensorPollingThread**(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

read_load_buffer()

reset_load_buffer()

terminate()

alive()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.loading.SerialDevice

Classes

SerialDevice(port[, baudrate, timeout, ...])	

Exceptions

SerialCommsException	Raised when the system receives a serial response it can't parse, likely a garbled line

class AFL.automation.loading.SerialDevice.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.SyringePump

Classes

SyringePump()	

class AFL.automation.loading.SyringePump.**SyringePump**

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

AFL.automation.loading.Tubing

Classes

Tubing(specid, length)	

class AFL.automation.loading.Tubing.**Tubing**(*specid*, *length*)

tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material': 'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in': 0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1517}]

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

AFL.automation.loading.TwoSelectorBlowoutSampleCell

Classes

Driver(name[, defaults, overrides])	
SampleCell()	
Tubing(specid, length)	
TwoSelectorBlowoutSampleCell(pump, selector, ...)	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**TwoSelectorBlowoutSampleCell**(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

setWasteLevel(*vol*)

AFL.automation.loading.UltimusVPressureController

Classes

PressureController()	Abstract superclass for pressure controllers that provides timed dispensing
UltimusVPressureController(port[, baud])	

class AFL.automation.loading.UltimusVPressureController.**UltimusVPressureController**(*port*, *baud=115200*)

__init__(*port*, *baud=115200*)

Initializes a DigitalOutPressureController

Params:

port (str): serial port to use

compute_checksum(*cmd*)

char_count(*cmd*)

package_cmd(*cmd*)

send_command(*cmd*)

set_P(*pressure*)

pressure: pressure to set in psi

AFL.automation.loading.ViciMultiposSelector

Classes

FlowSelector()	
SerialDevice(port[, baudrate, timeout, ...])	
ViciMultiposSelector(port[, baudrate, ...])	

class AFL.automation.loading.ViciMultiposSelector.**ViciMultiposSelector**(*port*, *baudrate=9600*, *portlabels=None*)

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*, *block=True*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

Parameters

- **port** (*String or int*) – the name, or number, of the port to switch to
- **direction** (*String, default=None*) – direction "CW" or "CCW" to perform the move in
- **block** (*bool, default=True*) – block return until move completes

getPort(*as_str=False*)

query the current selected position

AFL.automation.prepare

Functions

HD2OFactory(name[, phi_D2O, sld, properties])	Create a list of H2O/D2O solutions
compositionSweepFactory(name, components, ...)	
make_locs(slot, nrows, ncols)	
make_wellplate_locs(slot, size)	

Classes

ComponentDB([path])	
Deck()	
MassBalance()	
PipetteAction(source, dest, volume[, ...])	
Sample(name, target[, target_check, balancer])	
SampleSeries()	
Solute(name[, description, density, ...])	Specialization class for solute components
Solution(name, components[, properties])	
Solvent(name, density[, description, ...])	Specialization class for solute components

Modules

Component
DeckBuilderWidget
Dummy_OT2_Driver
OT2Client
PrepType
PrepareWidget
SampleSeriesWidget
StockBuilderWidget
SweepBuilderWidget
factory
utilities

AFL.automation.prepare.Component

Functions

enforce_units(value, unit_type)	Ensure that a number has units and convert to the default_units

Classes

Component(name, description)	Base class for all materials
PrepType(value[, names, module, qualname, ...])	

Exceptions

ParseException(pstr[, loc, msg, elem])	Exception thrown when a parse expression doesn't match the input string

class AFL.automation.prepare.Component.**Component**(*name*, *description*)

Base class for all materials

This class defines all of the basic properties and methods to be shared across material objects

__init__(*name*, *description*)

emit()

__hash__()

Needed so Components can be dictionary keys

copy()

__iter__()

Dummy iterator to mimic behavior of Mixture.

property mass

set_mass(*value*)

Setter for inline mass changes

property volume

set_volume(*value*)

Setter for inline volume changes

property density

property formula

property moles

property sld

property is_solute

property is_solvent

AFL.automation.prepare.DeckBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
Client([ip, port, username, interactive])	Communicate with APIServer
DeckBuilderWidget()	
DeckBuilderWidget_Model()	
DeckBuilderWidget_View()	
Dropdown(**kwargs)	Allows you to select a single item from a dropdown.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget**

__init__()

build_deck_object()

get_deck_config()

set_deck_config(*config*)

send_deck_cb(*event*)

save_cb(*event*)

load_cb(*event*)

update_deck_graphic_cb(*event*)

start()

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_Model**

__init__()

send_deck_config(*pi_ip='piot2'*, *align_script='/home/nistoroboto/align.py'*)

build_deck_object(*config*)

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_View**

__init__()

create_expanded_button(*description*, *button_style*)

draw_deck()

start()

AFL.automation.prepare.Dummy_OT2_Driver

Classes

Driver(name[, defaults, overrides])	
Dummy_OT2_Driver([overrides])	

class AFL.automation.prepare.Dummy_OT2_Driver.**Dummy_OT2_Driver**(*overrides=None*)

defaults = {'execute_delay': 1}

__init__(*overrides=None*)

reset_prep_targets()

add_prep_targets(*targets*, *reset=False*)

get_prep_target()

status()

reset_tipracks(*mount='both'*)

home(***kwargs*)

parse_well(*loc*)

get_wells(*locs*)

get_labware(*slot*)

load_labware(*name*, *slot*, *module=None*, ***kwargs*)

set_temp(*slot*, *temp*)
Set the temperature of a tempdeck in slot slot

get_temp(*slot*)
Get the temperature of a tempdeck in slot slot

deactivate_temp(*slot*)
Disablea tempdeck in slot slot

set_shake(*rpm*)

stop_shake()

set_shaker_temp(*temp*)

unlatch_shaker()

latch_shaker()

get_shaker_temp()

get_shake_rpm()

get_shake_latch_status()

load_instrument(*name*, *mount*, *tip_rack_slots*, ***kwargs*)

transfer(*source*, *dest*, *volume*, **args*, ***kwargs*)

set_aspirate_rate(*rate=150*)

set_dispense_rate(*rate=300*)

set_gantry_speed(*speed=400*)

AFL.automation.prepare.OT2Client

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
OT2Client([ip, port, username, interactive])	Communicate with AFL-automation server on OT-2
PipetteAction(source, dest, volume[, ...])	

class AFL.automation.prepare.OT2Client.**OT2Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Communicate with AFL-automation server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the AFL-automation server

transfer(*source*, *dest*, *volume*, *interactive=None*, ***kwargs*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str Source wells to transfer from. Wells should be specified as three*) – character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

reset_tipracks(*mount='both'*)

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

aspirate_rate(*rate*)

dispense_rate(*rate*)

home()

AFL.automation.prepare.PrepType

Functions

makeRegistar()	
prepRegistrar(prepType)	

Classes

Enum(value[, names, module, qualname, type, ...])	Create a collection of name/value pairs.
PrepType(value[, names, module, qualname, ...])	
auto([value])	Instances are replaced with an appropriate value in Enum class suites.

class AFL.automation.prepare.PrepType.**PrepType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

BaseComponent = 1

BaseMixture = 2

Solute = 3

Solvent = 4

Solution = 5

AFL.automation.prepare.PrepareWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
DeckBuilderWidget()	
Dropdown(**kwargs)	Allows you to select a single item from a dropdown.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
PrepareWidget()	
PrepareWidget_Model()	
PrepareWidget_View()	
SampleSeriesWidget(deck)	
StockBuilderWidget(deck)	
SweepBuilderWidget(deck)	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

class AFL.automation.prepare.PrepareWidget.**PrepareWidget**

__init__()

SweepBuilder_reset_cb(*event*)

StockBuilder_reset_cb(*event*)

SampleSeriesTool_reset_cb(*event*)

start()

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_Model**

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_View**

start(*deck_builder_widget*)

AFL.automation.prepare.SampleSeriesWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
Client([ip, port, username, interactive])	Communicate with APIServer
FloatText(**kwargs)	Displays a float value within a textbox.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
IntText(**kwargs)	Textbox widget that represents an integer.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
SampleSeriesWidget(deck)	
SampleSeriesWidget_Model(deck)	
SampleSeriesWidget_View()	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget**(*deck*)

__init__(*deck*)

apply_protocol_order_cb(*event*)

apply_protocol_order()

update_protocol_order_preview_cb(*event*)

make_protocol()

make_catch_protocol()

submit_cb(*event*)

build_label(*index*)

example_label_cb(*event*)

make_all_labels_cb(*event*)

make_all_labels()

sync_to_prepare_cb(*event*)

reset_uuid_cb(*event*)

update_sort_order_cb(*event*, *direction*)

make_mixing_wells()

update_mixing_well_preview_cb(*event*)

submit_mixing_wells_cb(*event*)

start()

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_Model**(*deck*)

__init__(*deck*)

adjust_protocol_order(*mix_order*)

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_View**

__init__()

make_component_grid(*components*, *nsamples*)

make_pipette_params(*name*)

make_mixing_wells()

start(*components*, *nsamples*, *stock_names*)

AFL.automation.prepare.StockBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

StockBuilderWidget(deck)
StockBuilderWidget_Model(deck)
StockBuilderWidget_View()

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget**(*deck*)

__init__(*deck*)

analyze_stocks_cb(*event*)

get_stock_objects()

add_stocks_to_deck()

get_stock_values()

save_cb(*event*, *pkl=True*)

load_cb(*event*)

update_location_check_cb(*event*)

make_stock_cb(*event*)

remove_stock_cb(*event*)

set_units_cb(*event*, *units*)

start()

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_Model**(*deck*)

__init__(*deck*)

add_stocks_to_deck(*all_stocks_dict*)

to_stock_objects(*all_stocks_dict*)

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_View**

__init__()

make_stock_tab(*stock_name*, *components*)

start()

AFL.automation.prepare.SweepBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
SweepBuilderWidget(deck)	
SweepBuilderWidget_Model(deck)	
SweepBuilderWidget_View()	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget**(*deck*)

__init__(*deck*)

plot_binary_cb(*click*)

plot_ternary_cb(*click*)

calc_sweep_cb(*click*)

validate_sweep_cb(*click*)

get_sweep_data()

get_deck()

update_component_row_cb(*event*)

start()

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_Model**(*deck*)

__init__(*deck*)

validate_sweep(*tolerance*, *progress=None*)

calc_sweep(*sweep_dict*, *progress*)

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_View**

__init__()

make_stock_grid(*component_names*)

make_ternary_plot(*component_names*)

make_binary_plot(*component_names*)

start(*component_names*)

AFL.automation.prepare.factory

Functions

HD2OFactory(name[, phi_D2O, sld, properties])	Create a list of H2O/D2O solutions
compositionSweepFactory(name, components, ...)	
has_units(value)	
is_concentration(value)	
is_mass(value)	
is_molarity(value)	
is_volume(value)	
listify(obj)	

Classes

Solution(name, components[, properties])	
product	product(*iterables, repeat=1) --> product object

AFL.automation.prepare.factory.**HD2OFactory**(*name*, *phi_D2O=None*, *sld=None*, *properties=None*)

Create a list of H2O/D2O solutions

AFL.automation.prepare.factory.**compositionSweepFactory**(*name*, *components*, *vary_components*, *lo*, *hi*, *num*, *logspace=False*, *properties=None*, *progress=None*)

AFL.automation.prepare.utilities

Functions

make_locs(slot, nrows, ncols)	
make_wellplate_locs(slot, size)	

AFL.automation.prepare.utilities.**make_locs**(*slot*, *nrows*, *ncols*)

AFL.automation.prepare.utilities.**make_wellplate_locs**(*slot*, *size*)

AFL.automation.sample_env

Modules

TemperatureDeck	

AFL.automation.sample_env.TemperatureDeck

Classes

Driver(name[, defaults, overrides])	
TemperatureDeck([overrides])	

class AFL.automation.sample_env.TemperatureDeck.**TemperatureDeck**(*overrides=None*)

defaults = {'serial_port': '/dev/ttyACM0', 'temperature_move_sleep': 0.2, 'temperature_move_timeout': 900}

__init__(*overrides=None*)

set_temp(*sp*)

move_temp(*temperature*, *wait=30*, *tolerance=0.1*)

status()

read_temp()

AFL.automation.shared

Modules

DataLabelerWidget
DatasetWidget
DiffractionLabeler
MutableQueue
PersistentConfig
ServerDiscovery
exceptions
serialization
units
utilities
widgetui

AFL.automation.shared.DataLabelerWidget

Functions

sqrt(x, /)	Return the square root of x.

Classes

DataLabelerModel(dataset)	
DataLabelerView()	
DataLabelerWidget(input_dataset, ...[, ...])	
OrdinalEncoder(*[, categories, dtype, ...])	Encode categorical features as an integer array.

class AFL.automation.shared.DataLabelerWidget.**DataLabelerWidget**(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

__init__(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

Parameters

- **dataset** (*xr.Dataset*) – Dataset from AFL
- **sas_variable** (*str*) – Name of data variable in the *xarray.Dataset* that holds the scattering data
- **composition_variable** (*str | List[str]*) – Name of data variable in the *xarray.Dataset* that holds the composition. If the composition is split across multiple variables, pass in a list of variables.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to each sample or measurement. This is typically named 'sample' in much of the AFL agent codebase.
- **fit_variable** (*Optional[str]*) – If not none, this data will be plotted along with the sas_variable data. This data variable should have the same shape as sas_variable.

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

composition_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

class AFL.automation.shared.DataLabelerWidget.**DataLabelerModel**(*dataset: Dataset*)

__init__(*dataset: Dataset*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

class AFL.automation.shared.DataLabelerWidget.**DataLabelerView**

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_composition_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *ternary*, *components*)

AFL.automation.shared.DatasetWidget

Classes

DatasetWidget(dataset[, sample_dim, ...])	
DatasetWidget_Model(dataset, sample_dim)	
DatasetWidget_View([...])	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class AFL.automation.shared.DatasetWidget.**DatasetWidget**(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

__init__(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

Interactive widget for viewing compositionally varying scattering data

Parameters

- **dataset** (*xr.Dataset*) – *xarray.Dataset* containing scattering data and compositions to be plotted.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to sample variation, typically “sample”
- **comps_variable** (*Optional[str]*) – The name of the *xarray* variable to plot as compositional data. Optional, if not specified, can be customized in the GUI.

  Only the first two columns of the data will be used in the plot. If the compositions are in separate `xarray.DataArray`s, they should be grouped into single `xarray.DataArray`s like so:

  ```
  `python ds['comps'] = ds[['A','B','C']].to_array('component').
  transpose(...,'component') `
  ```
- **comps_color_variable** (*Optional[str]*) – The name of the *xarray* variable to use as the colorscale of the compositional data scatter plot. Optional, if not specified, can be customized in the GUI.
- **xmin** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **xmax** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **Usage**
- **-----**
- **```python** –
- **DatasetWidget(ds)** (*widget =*)

- widget.run()
- ``` _

next_button_callback(**args*)

prev_button_callback(**args*)

goto_callback(**args*)

composition_click_callback(*figure*, *location*, *click*)

update_composition_plot()

update_scattering_plot()

update_plots()

get_comps()

initialize_plots(**args*)

update_colors(**args*)

apply_sel(**args*)

apply_isel(**args*)

extract_var(**args*)

combine_vars(**args*)

reset_dataset(**args*)

update_dropdowns(**args*)

update_sample_dim(**args*)

update_extract_coords(*change*)

run()

class AFL.automation.shared.DatasetWidget.DatasetWidget_Model(*dataset: Dataset*, *sample_dim: str*)

__init__(*dataset: Dataset*, *sample_dim: str*)

property dataset

reset_dataset()

split_vars()

Heuristically try to split vars into categories

get_non_sample_dims(*var: str*)

apply_sel(*kw*)

apply_isel(*kw*)

combine_vars(*combined_var: str*, *to_combine_vars: List[str]*)

extract_var(*extract_from_var: str*, *extract_from_coord: str*)

get_composition(*variable*)

get_scattering(*variable*, *index*)

class AFL.automation.shared.DatasetWidget.**DatasetWidget_View**(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

__init__(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

update_colorscale(*colors=None*)

update_selected(***kw*)

update_dropdowns(*sample_vars=None*, *scatt_vars=None*, *comp_vars=None*)

plot_sas(*x*, *y*, *name='SAS'*, *append=False*)

plot_comp(*x*, *y*, *z=None*, *xname='x'*, *yname='y'*, *zname='z'*, *colors=None*)

init_plots()

init_buttons()

init_checkboxes()

init_dropdowns()

init_inputs()

run()

AFL.automation.shared.DiffractionLabeler

Functions

sqrt(x, /)	Return the square root of x.

Classes

DiffractionLabeler(saxs_data, ...)	
DiffractionLabelerModel(saxs_data, ...)	
DiffractionLabelerView()	
OrdinalEncoder(*[, categories, dtype, ...])	Encode categorical features as an integer array.

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabeler**(*saxs_data*, *composition_data*, *possible_phase_labels*)

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

ternary_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerModel**(*saxs_data*, *composition_data*, *possible_phase_labels*)

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerView**

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_ternary_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *possible_phase_labels*)

AFL.automation.shared.MutableQueue

Classes

MutableQueue()	Thread-safe, mutable queue

Exceptions

Empty	Exception raised by Queue.get(block=0)/get_nowait().
Full	Exception raised by Queue.put(block=0)/put_nowait().

class AFL.automation.shared.MutableQueue.**MutableQueue**

Thread-safe, mutable queue

Unlike the standard library CPython queue, this class supportes positional inserts, deletions, and reordering. The tradeoff is performance for both reads and writes to the queue.

__init__()

qsize()

iterationid()

empty()

put(*item*, *loc*)

Insert an item at the top of the queue

remove(*loc*)

Remove an item from the queue

get(*loc=0*, *block=True*, *timeout=None*)

Get next item from queue

move(*old_index*, *new_index=None*)

Move item in queue

AFL.automation.shared.PersistentConfig

Classes

MutableMapping()	A MutableMapping is a generic container for associating key/value pairs.
PersistentConfig(path[, defaults, ...])	A dictionary-like class that serializes changes to disk

class AFL.automation.shared.PersistentConfig.**PersistentConfig**(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

A dictionary-like class that serializes changes to disk

This class provides dictionary-like setters and getters (e.g., [] and .update()) but, by default, all modifications are written into a file in json format with the root keys as timestamps. Modifications can be blocked by locking the config, and writing to disk can be disabled by setting the appropriate member attributes (see constructor). On instantiation, if provided with a previously saved configuration file, PersistentConfig will load the file's contents into memory and use the more recent configuration.

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

__getitem__(*key*)

Dictionary-like getter via config["param"]

__setitem__(*key*, *value*)

Dictionary-like setter via config["param"]=value

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

toJSON()

Serialize the config to json

update(*update_dict*)

Update several values in config at once

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

revert(*nth=None*, *datetime_key=None*)

Revert config to a historical config

Parameters

- **nth** (int, **optional***) – Integer index of historical value to revert to. Can be negative to count from end of history. Note that -1 will correspond to the current config.
- **datetime_key** (str, **optional**) – datetime formatted string as defined by datetime_key_format

get_historical_values(*key*, *convert_to_datetime=False*)

Convenience method for gathering historical values of a parameter

AFL.automation.shared.ServerDiscovery

Classes

AsyncServiceBrowser(zeroconf, type_[, ...])	Used to browse for a service for specific type(s).
AsyncServiceInfo	An async version of ServiceInfo.
AsyncZeroconf([interfaces, unicast, ...])	Implementation of Zeroconf Multicast DNS Service Discovery
AsyncZeroconfServiceTypes()	An async version of ZeroconfServiceTypes.
IPVersion(value[, names, module, qualname, ...])	
RunThread(coro)	
ServerDiscovery()	ServerDiscovery class
ServiceBrowser(zc, type_[, handlers, ...])	Used to browse for a service of a specific type.
ServiceInfo	Service information.
ServiceStateChange(value[, names, module, ...])	
Zeroconf([interfaces, unicast, ip_version, ...])	Implementation of Zeroconf Multicast DNS Service Discovery

class AFL.automation.shared.ServerDiscovery.**RunThread**(*coro*)

__init__(*coro*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.shared.ServerDiscovery.**ServerDiscovery**

ServerDiscovery class

This class is used to discover AFL servers on the network using zeroconf requests that match a particular specification.

__init__()

on_service_state_change(*zeroconf: Zeroconf*, *service_type: str*, *name: str*, *state_change: ServiceStateChange*) → None

async manage_service_info_to_list(*zeroconf*, *service_type*, *name*, *append_or_remove*)

async get_service_info(*zeroconf: Zeroconf*, *service_type: str*, *name: str*) → None

async aio_find_server_by_name(*service_name*)

discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

async classmethod sa_aio_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

classmethod sa_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_name(*service_name*)

Disambiguator for either matching or discovering a specific server by name

match_server_by_name(*service_name*)

Looks through the registry of discovered services for an exact name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_partial_name(*service_name*)

Looks through the registry of discovered services for a partial name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_property_match(*property_name*, *property_value*)

Looks through the registry of discovered services for a partial match in a property string. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

AFL.automation.shared.exceptions

Exceptions

EmptyException	Raised when a mixture runs out of mass or volume
MixingException	Raised when a mixture cannot be made
NoDeviceFoundException	Raised when no matching device can be found on the selected port
NotFoundError	
SerialCommsException	Raised when the system receives a serial response it can't parse, likely a garbled line

exception AFL.automation.shared.exceptions.**EmptyException**

Raised when a mixture runs out of mass or volume

exception AFL.automation.shared.exceptions.**MixingException**

Raised when a mixture cannot be made

exception AFL.automation.shared.exceptions.**SerialCommsException**

Raised when the system receives a serial response it can't parse, likely a garbled line

exception AFL.automation.shared.exceptions.**NoDeviceFoundException**

Raised when no matching device can be found on the selected port

exception AFL.automation.shared.exceptions.**NotFoundError**

AFL.automation.shared.serialization

Functions

deserialize(pickled_str)	
is_serialized(obj)	
serialize(obj)	

Exceptions

UnpicklingError	

AFL.automation.shared.serialization.**serialize**(*obj*)

AFL.automation.shared.serialization.**is_serialized**(*obj*)

AFL.automation.shared.serialization.**deserialize**(*pickled_str*)

AFL.automation.shared.units

Functions

enforce_units(value, unit_type)	Ensure that a number has units and convert to the default_units
get_unit_type(value)	
has_units(value)	
is_concentration(value)	
is_density(value)	
is_mass(value)	
is_molarity(value)	
is_volume(value)	

AFL.automation.shared.units.**has_units**(*value*)

AFL.automation.shared.units.**is_volume**(*value*)

AFL.automation.shared.units.**is_molarity**(*value*)

AFL.automation.shared.units.**is_mass**(*value*)

AFL.automation.shared.units.**is_density**(*value*)

AFL.automation.shared.units.**is_concentration**(*value*)

AFL.automation.shared.units.**get_unit_type**(*value*)

AFL.automation.shared.units.**enforce_units**(*value*, *unit_type*)

Ensure that a number has units and convert to the default_units

AFL.automation.shared.utilities

Functions

has_units(value)	
listify(obj)	
makeRegistar()	
mpl_plot_to_bytes([fig, format])	
tprint(in_str)	
xarray_to_bytes(ds[, format])	

AFL.automation.shared.utilities.**listify**(*obj*)

AFL.automation.shared.utilities.**tprint**(*in_str*)

AFL.automation.shared.utilities.**makeRegistar**()

AFL.automation.shared.utilities.**mpl_plot_to_bytes**(*fig=None*, *format='svg'*)

AFL.automation.shared.utilities.**xarray_to_bytes**(*ds*, *format='svg'*)

AFL.automation.shared.widgetui

Functions

clear_output([wait])	Clear the output of the current cell receiving output.
client_construct_ui(self[, return_ui, ...])	
display(*objs[, include, exclude, metadata, ...])	Display a Python object in all frontends.

Classes

Markdown([data, url, filename, metadata])	

AFL.automation.shared.widgetui.**client_construct_ui**(*self*, *return_ui=False*, *display_ui=True*)

6.1.2 AFL.automation.sample

Modules

CastingServer	

AFL.automation.sample.CastingServer

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

Classes

CastingServer(prep_url[, overrides])	
Client([ip, port, username, interactive])	Communicate with APIServer
Driver(name[, defaults, overrides])	
OT2Client([ip, port, username, interactive])	Communicate with AFL-automation server on OT-2

class AFL.automation.sample.CastingServer.**CastingServer**(*prep_url*, *overrides=None*)

defaults = {'manifest': '/home/afl642/', 'manifest_cast': '/home/afl642/', 'manifest_prep': '/home/afl642/'}

__init__(*prep_url*, *overrides=None*)

init_casting_manifest(*attrs=None*, *overwrite=False*)

status()

update_status(*value*)

log_event(*manifest_key*, *event*, *write=True*)

assign_targets(*protocols*, *target_map*)

prepare_casting_stocks(***spec*)

Spec should contain sample_name and a protocol

bulk_cast_films(**spec*)

Spec should contain sample_name and a protocol

prepare_and_cast(**sample*)

6.2 Modules

This page provides an overview of the main modules in the AFL-automation framework.

6.2.1 API Server

The APIServer module provides HTTP microservices with authentication, task queueing, and UI generation.

AFL.automation.APIServer

AFL.automation.APIServer

Modules

APIServer
Client
Driver
DummyDriver
DummyOT2Driver
LoggerFilter
QueueDaemon

AFL.automation.APIServer.APIServer

Functions

create_access_token(identity[, fresh, ...])	Create a new access token.
create_refresh_token(identity[, ...])	Create a new refresh token.
get_jwt_identity()	In a protected endpoint, this will return the identity of the JWT that is accessing the endpoint.
jsonify(*args, **kwargs)	Serialize the given arguments as JSON, and return a Response object with the application/json mimetype.
jwt_required([optional, fresh, refresh, ...])	A decorator to protect a Flask endpoint with JSON Web Tokens.
listify(obj)	
render_template(template_name_or_list, **context)	Render a template by name with the given context.
send_file(path_or_file[, mimetype, ...])	Send the contents of a file to the client.
set_access_cookies(response, ...[, max_age, ...])	Modifiy a Flask Response to set a cookie containing the access JWT.
set_refresh_cookies(response, ...[, ...])	Modifiy a Flask Response to set a cookie containing the refresh JWT.
strtobool(val)	Convert a string representation of truth to true (1) or false (0).
unset_jwt_cookies(response[, domain])	Modifiy a Flask Response to delete the cookies containing access or refresh JWTs.

AFL.automation.APIServer.APIServer.create_access_token

AFL.automation.APIServer.APIServer.**create_access_token**(*identity: Any*, *fresh: bool | float | timedelta = False*, *expires_delta: Literal[False] | timedelta | None = None*, *additional_claims=None*, *additional_headers=None*)

Create a new access token.

Parameters

- **identity** – The identity of this token. This must either be a string, or you must have defined `user_identity_loader()` in order to convert the object you passed in into a string.
- **fresh** – If this token should be marked as fresh, and can thus access endpoints protected with `@jwt_required(fresh=True)`. Defaults to `False`.

  This value can also be a `datetime.timedelta`, which indicate how long this token will be considered fresh.
- **expires_delta** – A `datetime.timedelta` for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the `JWT_ACCESS_TOKEN_EXPIRES` config value (see Configuration Options)
- **additional_claims** – Optional. A hash of claims to include in the access token. These claims are merged into the default claims (exp, iat, etc) and claims returned from the `additional_claims_loader()` callback. On conflict, these claims take precedence.
- **headers** – Optional. A hash of headers to include in the access token. These headers are merged into the default headers (alg, typ) and headers returned from the `additional_headers_loader()` callback. On conflict, these headers take precedence.

Returns
An encoded access token

AFL.automation.APIServer.APIServer.create_refresh_token

AFL.automation.APIServer.APIServer.**create_refresh_token**(*identity: Any*, *expires_delta: Literal[False] | timedelta | None = None*, *additional_claims=None*, *additional_headers=None*)

Create a new refresh token.

Parameters

- **identity** – The identity of this token. This must either be a string, or you must have defined `user_identity_loader()` in order to convert the object you passed in into a string.
- **expires_delta** – A `datetime.timedelta` for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the `JWT_REFRESH_TOKEN_EXPIRES` config value (see Configuration Options)
- **additional_claims** – Optional. A hash of claims to include in the refresh token. These claims are merged into the default claims (exp, iat, etc) and claims returned from the `additional_claims_loader()` callback. On conflict, these claims take precedence.
- **headers** – Optional. A hash of headers to include in the refresh token. These headers are merged into the default headers (alg, typ) and headers returned from the `additional_headers_loader()` callback. On conflict, these headers take precedence.

Returns
An encoded refresh token

AFL.automation.APIServer.APIServer.get_jwt_identity

AFL.automation.APIServer.APIServer.**get_jwt_identity**() → Any

In a protected endpoint, this will return the identity of the JWT that is accessing the endpoint. If no JWT is present due to `jwt_required(optional=True)`, `None` is returned.

Returns
The identity of the JWT in the current request

AFL.automation.APIServer.APIServer.jsonify

AFL.automation.APIServer.APIServer.**jsonify**(**args: t.Any*, ***kwargs: t.Any*) → Response

Serialize the given arguments as JSON, and return a `Response` object with the `application/json` mimetype. A dict or list returned from a view will be converted to a JSON response automatically without needing to call this.

This requires an active request or application context, and calls `app.json.response()`.

In debug mode, the output is formatted with indentation to make it easier to read. This may also be controlled by the provider.

Either positional or keyword arguments can be given, not both. If no arguments are given, `None` is serialized.

Parameters

- **args** – A single value to serialize, or multiple values to treat as a list to serialize.
- **kwargs** – Treat as a dict to serialize.

Changed in version 2.2: Calls `current_app.json.response`, allowing an app to override the behavior.

Changed in version 2.0.2: `decimal.Decimal` is supported by converting to a string.

Changed in version 0.11: Added support for serializing top-level arrays. This was a security risk in ancient browsers. See security-json.

Added in version 0.2.

AFL.automation.APIServer.APIServer.jwt_required

AFL.automation.APIServer.APIServer.**jwt_required**(*optional: bool = False*, *fresh: bool = False*, *refresh: bool = False*, *locations: str | Sequence | None = None*, *verify_type: bool = True*, *skip_revocation_check: bool = False*) → Any

A decorator to protect a Flask endpoint with JSON Web Tokens.

Any route decorated with this will require a valid JWT to be present in the request (unless optional=True, in which case no JWT is also valid) before the endpoint can be called.

Parameters

- **optional** – If `True`, allow the decorated endpoint to be accessed if no JWT is present in the request. Defaults to `False`.
- **fresh** – If `True`, require a JWT marked with `fresh` to be able to access this endpoint. Defaults to `False`.
- **refresh** – If `True`, requires a refresh JWT to access this endpoint. If `False`, requires an access JWT to access this endpoint. Defaults to `False`.
- **locations** – A location or list of locations to look for the JWT in this request, for example `'headers'` or `['headers', 'cookies']`. Defaults to `None` which indicates that JWTs will be looked for in the locations defined by the `JWT_TOKEN_LOCATION` configuration option.
- **verify_type** – If `True`, the token type (access or refresh) will be checked according to the `refresh` argument. If `False`, type will not be checked and both access and refresh tokens will be accepted.
- **skip_revocation_check** – If `True`, revocation status of the token will be *not* checked. If `False`, revocation status of the token will be checked.

AFL.automation.APIServer.APIServer.listify

AFL.automation.APIServer.APIServer.**listify**(*obj*)

AFL.automation.APIServer.APIServer.render_template

AFL.automation.APIServer.APIServer.**render_template**(*template_name_or_list: str | Template | List[str | Template]*, ***context: Any*) → str

Render a template by name with the given context.

Parameters

- **template_name_or_list** – The name of the template to render. If a list is given, the first name to exist will be rendered.
- **context** – The variables to make available in the template.

AFL.automation.APIServer.APIServer.send_file

AFL.automation.APIServer.APIServer.**send_file**(*path_or_file: PathLike | str | BinaryIO*, *mimetype: str | None = None*, *as_attachment: bool = False*, *download_name: str | None = None*, *conditional: bool = True*, *etag: bool | str = True*, *last_modified: datetime | int | float | None = None*, *max_age: int | Callable[[str | None], int | None] | None = None*) → Response

Send the contents of a file to the client.

The first argument can be a file path or a file-like object. Paths are preferred in most cases because Werkzeug can manage the file and get extra information from the path. Passing a file-like object requires that the file is opened in binary mode, and is mostly useful when building a file in memory with `io.BytesIO`.

Never pass file paths provided by a user. The path is assumed to be trusted, so a user could craft a path to access a file you didn't intend. Use `send_from_directory()` to safely serve user-requested paths from within a directory.

If the WSGI server sets a `file_wrapper` in `environ`, it is used, otherwise Werkzeug's built-in wrapper is used. Alternatively, if the HTTP server supports `X-Sendfile`, configuring Flask with `USE_X_SENDFILE = True` will tell the server to send the given path, which is much more efficient than reading it in Python.

Parameters

- **path_or_file** – The path to the file to send, relative to the current working directory if a relative path is given. Alternatively, a file-like object opened in binary mode. Make sure the file pointer is seeked to the start of the data.
- **mimetype** – The MIME type to send for the file. If not provided, it will try to detect it from the file name.
- **as_attachment** – Indicate to a browser that it should offer to save the file instead of displaying it.
- **download_name** – The default name browsers will use when saving the file. Defaults to the passed file name.
- **conditional** – Enable conditional and range responses based on request headers. Requires passing a file path and `environ`.
- **etag** – Calculate an ETag for the file, which requires passing a file path. Can also be a string to use instead.
- **last_modified** – The last modified time to send for the file, in seconds. If not provided, it will try to detect it from the file path.
- **max_age** – How long the client should cache the file, in seconds. If set, `Cache-Control` will be `public`, otherwise it will be `no-cache` to prefer conditional caching.

Changed in version 2.0: `download_name` replaces the `attachment_filename` parameter. If `as_attachment=False`, it is passed with `Content-Disposition: inline` instead.

Changed in version 2.0: `max_age` replaces the `cache_timeout` parameter. `conditional` is enabled and `max_age` is not set by default.

Changed in version 2.0: `etag` replaces the `add_etags` parameter. It can be a string to use instead of generating one.

Changed in version 2.0: Passing a file-like object that inherits from `TextIOBase` will raise a `ValueError` rather than sending an empty file.

Added in version 2.0: Moved the implementation to Werkzeug. This is now a wrapper to pass some Flask-specific arguments.

Changed in version 1.1: `filename` may be a `PathLike` object.

Changed in version 1.1: Passing a `BytesIO` object supports range requests.

Changed in version 1.0.3: Filenames are encoded with ASCII instead of Latin-1 for broader compatibility with WSGI servers.

Changed in version 1.0: UTF-8 filenames as specified in **RFC 2231** are supported.

Changed in version 0.12: The filename is no longer automatically inferred from file objects. If you want to use automatic MIME and etag support, pass a filename via `filename_or_fp` or `attachment_filename`.

Changed in version 0.12: `attachment_filename` is preferred over `filename` for MIME detection.

Changed in version 0.9: `cache_timeout` defaults to *`Flask.get_send_file_max_age()`*.

Changed in version 0.7: MIME guessing and etag support for file-like objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself.

Changed in version 0.5: The `add_etags`, `cache_timeout` and `conditional` parameters were added. The default behavior is to add etags.

Added in version 0.2.

AFL.automation.APIServer.APIServer.set_access_cookies

`AFL.automation.APIServer.APIServer.`**`set_access_cookies`**(*response: Response*, *encoded_access_token: str*, *max_age=None*, *domain=None*) → None

Modifiy a Flask Response to set a cookie containing the access JWT. Also sets the corresponding CSRF cookies if `JWT_CSRF_IN_COOKIES` is `True` (see Configuration Options)

Parameters

- **`response`** – A Flask Response object.
- **`encoded_access_token`** – The encoded access token to set in the cookies.
- **`max_age`** – The max age of the cookie. If this is None, it will use the `JWT_SESSION_COOKIE` option (see Configuration Options). Otherwise, it will use this as the cookies `max-age` and the JWT_SESSION_COOKIE option will be ignored. Values should be the number of seconds (as an integer).
- **`domain`** – The domain of the cookie. If this is None, it will use the `JWT_COOKIE_DOMAIN` option (see Configuration Options). Otherwise, it will use this as the cookies `domain` and the JWT_COOKIE_DOMAIN option will be ignored.

AFL.automation.APIServer.APIServer.set_refresh_cookies

`AFL.automation.APIServer.APIServer.`**`set_refresh_cookies`**(*response: Response*, *encoded_refresh_token: str*, *max_age: int | None = None*, *domain: str | None = None*) → None

Modifiy a Flask Response to set a cookie containing the refresh JWT. Also sets the corresponding CSRF cookies if `JWT_CSRF_IN_COOKIES` is `True` (see Configuration Options)

Parameters

- **`response`** – A Flask Response object.
- **`encoded_refresh_token`** – The encoded refresh token to set in the cookies.

- **max_age** – The max age of the cookie. If this is None, it will use the JWT_SESSION_COOKIE option (see Configuration Options). Otherwise, it will use this as the cookies max-age and the JWT_SESSION_COOKIE option will be ignored. Values should be the number of seconds (as an integer).
- **domain** – The domain of the cookie. If this is None, it will use the JWT_COOKIE_DOMAIN option (see Configuration Options). Otherwise, it will use this as the cookies domain and the JWT_COOKIE_DOMAIN option will be ignored.

AFL.automation.APIServer.APIServer.strtobool

AFL.automation.APIServer.APIServer.**strtobool**(*val*)

Convert a string representation of truth to true (1) or false (0).

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'val' is anything else.

AFL.automation.APIServer.APIServer.unset_jwt_cookies

AFL.automation.APIServer.APIServer.**unset_jwt_cookies**(*response: Response*, *domain: str | None = None*) → None

Modifiy a Flask Response to delete the cookies containing access or refresh JWTs. Also deletes the corresponding CSRF cookies if applicable.

Parameters

response – A Flask Response object

Classes

APIServer(name[, data, experiment, contact, ...])	
CORS([app])	Initializes Cross Origin Resource sharing for the application.
FileHandler(filename[, mode, encoding, ...])	A handler class which writes formatted logging records to disk files.
Flask(import_name[, static_url_path, ...])	The flask object implements a WSGI application and acts as the central object.
IPVersion(value[, names, module, qualname, ...])	
JWTManager([app, add_context_processor])	An object used to hold JWT settings and callback functions for the Flask-JWT-Extended extension.
LoggerFilter(*filters)	
MutableQueue()	Thread-safe, mutable queue
QueueDaemon(app, driver, task_queue, history)	
SMTPHandler(mailhost, fromaddr, toaddrs, subject)	A handler class which sends an SMTP email for each logging event.
ServiceInfo	Service information.
Zeroconf([interfaces, unicast, ip_version, ...])	Implementation of Zeroconf Multicast DNS Service Discovery

AFL.automation.APIServer.APIServer.APIServer

class AFL.automation.APIServer.APIServer.**APIServer**(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

Bases: object

__init__(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

Methods

__init__(name[, data, experiment, contact, ...])	
add_standard_routes()	
add_unqueued_routes()	
advertise_zeroconf(**kwargs)	
clear_history()	
clear_queue()	
create_queue(driver[, add_unqueued])	
debug()	
deposit_obj()	Store an object named obj in the driver's dropbox If a uuid is provided, the object will be stored with that uuid Otherwise, a new uuid will be generated.
driver_status()	
enqueue()	
get_driver_object()	
get_info()	Live, status page of the robot
get_queue()	
get_queue_iteration()	
get_queued_commands()	
get_quickbar()	Return the functions, params, and defaults to be shown in this server's quickbar
get_server_time()	
get_unqueued_commands()	

continues on next page

Table 1 – continued from previous page

halt()	
index()	Live, status page of the robot
index_new()	Live, status page of the robot
init()	
init_logging([toaddrs])	
is_server_live()	
login()	
login_test()	
move_item()	
pause()	
query_driver()	
queue_state()	
remove_item()	
remove_items()	
render_unqueued(func, kwargs_add, **kwargs)	Convert an unqueued return item into web-suitable output
reorder_queue()	
reset_queue_daemon([driver])	
retrieve_obj()	Retrieve an object from the driver's dropbox uuid specifies the object to retrieve delete specifies whether to delete the object after retrieval
run(**kwargs)	
run_threaded([start_thread])	
send_1d_plot(result[, multi])	
send_array_as_jpg(array[, log_image, ...])	
set_driver_object()	
webapp()	Live, status page of the robot

__init__(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

create_queue(*driver*, *add_unqueued=True*)

reset_queue_daemon(*driver=None*)

advertise_zeroconf(***kwargs*)

run(***kwargs*)

run_threaded(*start_thread=True*, ***kwargs*)

add_standard_routes()

get_info()
Live, status page of the robot

get_quickbar()
Return the functions, params, and defaults to be shown in this server's quickbar

is_server_live()

get_unqueued_commands()

get_queued_commands()

add_unqueued_routes()

query_driver()

init_logging(*toaddrs=None*)

index()
Live, status page of the robot

index_new()
Live, status page of the robot

webapp()
Live, status page of the robot

render_unqueued(*func*, *kwargs_add*, ***kwargs*)
Convert an unqueued return item into web-suitable output

send_1d_plot(*result*, *multi=False*, ***kwargs*)

send_array_as_jpg(*array*, *log_image=False*, *max_val=None*, *fillna=0.0*, ***kwargs*)

queue_state()

driver_status()

get_queue()

get_queue_iteration()

deposit_obj()
Store an object named obj in the driver's dropbox If a uuid is provided, the object will be stored with that uuid Otherwise, a new uuid will be generated. In either case, the uuid will be returned to the client.

retrieve_obj()
Retrieve an object from the driver's dropbox uuid specifies the object to retrieve delete specifies whether to delete the object after retrieval

set_driver_object()

get_driver_object()

enqueue()

reorder_queue()

remove_items()

remove_item()

move_item()

clear_queue()

clear_history()

debug()

pause()

halt()

init()

login()

get_server_time()

login_test()

AFL.automation.APIServer.APIServer.CORS

class AFL.automation.APIServer.APIServer.**CORS**(*app=None*, ***kwargs*)

Bases: object

Initializes Cross Origin Resource sharing for the application. The arguments are identical to *cross_origin*, with the addition of a *resources* parameter. The resources parameter defines a series of regular expressions for resource paths to match and optionally, the associated options to be applied to the particular resource. These options are identical to the arguments to *cross_origin*.

The settings for CORS are determined in the following order

1. Resource level settings (e.g when passed as a dictionary)
2. Keyword argument settings
3. App level configuration settings (e.g. CORS_*)
4. Default settings

Note: as it is possible for multiple regular expressions to match a resource path, the regular expressions are first sorted by length, from longest to shortest, in order to attempt to match the most specific regular expression. This allows the definition of a number of specific resource options, with a wildcard fallback for all other resources.

Parameters

- **resources** (*dict, iterable or string*) – The series of regular expression and (optionally) associated CORS options to be applied to the given resource path.

  If the argument is a dictionary, it's keys must be regular expressions, and the values must be a dictionary of kwargs, identical to the kwargs of this function.

If the argument is a list, it is expected to be a list of regular expressions, for which the app-wide configured options are applied.

If the argument is a string, it is expected to be a regular expression for which the app-wide configured options are applied.

Default : Match all and apply app-level configuration

- **origins** (*list, string or regex*) – The origin, or list of origins to allow requests from. The origin(s) may be regular expressions, case-sensitive strings, or else an asterisk.

  Note

  origins must include the schema and the port (if not port 80), e.g., *CORS(app, origins=["http://localhost:8000", "https://example.com"])*.

  Default : '*'

- **methods** (*list or string*) – The method or list of methods which the allowed origins are allowed to access for non-simple requests.

  Default : [GET, HEAD, POST, OPTIONS, PUT, PATCH, DELETE]

- **expose_headers** (*list or string*) – The header or list which are safe to expose to the API of a CORS API specification.

  Default : None

- **allow_headers** (*list, string or regex*) – The header or list of header field names which can be used when this resource is accessed by allowed origins. The header(s) may be regular expressions, case-sensitive strings, or else an asterisk.

  Default : '*', allow all headers

- **supports_credentials** (*bool*) – Allows users to make authenticated requests. If true, injects the *Access-Control-Allow-Credentials* header in responses. This allows cookies and credentials to be submitted across domains.

  note
  This option cannot be used in conjunction with a '*' origin

  Default : False

- **max_age** (*timedelta, integer, string or None*) – The maximum time for which this CORS request maybe cached. This value is set as the *Access-Control-Max-Age* header.

  Default : None

- **send_wildcard** (*bool*) – If True, and the origins parameter is *, a wildcard *Access-Control-Allow-Origin* header is sent, rather than the request's *Origin* header.

  Default : False

- **vary_header** (*bool*) – If True, the header Vary: Origin will be returned as per the W3 implementation guidelines.

  Setting this header when the *Access-Control-Allow-Origin* is dynamically generated (e.g. when there is more than one allowed origin, and an Origin than '*' is returned) informs CDNs and other caches that the CORS headers are dynamic, and cannot be cached.

  If False, the Vary header will never be injected or altered.

  Default : True

- **allow_private_network** (*bool*) – If True, the response header *Access-Control-Allow-Private-Network* will be set with the value 'true' whenever the request header *Access-Control-Request-Private-Network* has a value 'true'.

  If False, the response header *Access-Control-Allow-Private-Network* will be set with the value 'false' whenever the request header *Access-Control-Request-Private-Network* has a value of 'true'.

  If the request header *Access-Control-Request-Private-Network* is not present or has a value other than 'true', the response header *Access-Control-Allow-Private-Network* will not be set.

  Default : True

__init__(*app=None*, ***kwargs*)

Methods

__init__([app])
init_app(app, **kwargs)

__init__(*app=None*, ***kwargs*)

init_app(*app*, ***kwargs*)

AFL.automation.APIServer.APIServer.FileHandler

class AFL.automation.APIServer.APIServer.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Bases: StreamHandler

A handler class which writes formatted logging records to disk files.

__init__(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Open the specified file and use it as the stream for logging.

Methods

__init__(filename[, mode, encoding, delay, ...])	Open the specified file and use it as the stream for logging.
acquire()	Acquire the I/O thread lock.
addFilter(filter)	Add the specified filter to this handler.
close()	Closes the stream.
createLock()	Acquire a thread lock for serializing access to the underlying I/O.
emit(record)	Emit a record.
filter(record)	Determine if a record is loggable by consulting all the filters.
flush()	Flushes the stream.
format(record)	Format the specified record.
get_name()	
handle(record)	Conditionally emit the specified logging record.
handleError(record)	Handle errors which occur during an emit() call.
release()	Release the I/O thread lock.
removeFilter(filter)	Remove the specified filter from this handler.
setFormatter(fmt)	Set the formatter for this handler.
setLevel(level)	Set the logging level of this handler.
setStream(stream)	Sets the StreamHandler's stream to the specified value, if it is different.
set_name(name)	

Attributes

name
terminator

__init__(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Open the specified file and use it as the stream for logging.

close()

Closes the stream.

emit(*record*)

Emit a record.

If the stream was not opened because 'delay' was specified in the constructor, open it before calling the superclass's emit.

If stream is not open, current mode is 'w' and *_closed=True*, record will not be emitted (see Issue #42378).

acquire()

Acquire the I/O thread lock.

addFilter(*filter*)

Add the specified filter to this handler.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()

Flushes the stream.

format(*record*)

Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(*record*)

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

property name

release()

Release the I/O thread lock.

removeFilter(*filter*)

Remove the specified filter from this handler.

setFormatter(*fmt*)

Set the formatter for this handler.

setLevel(*level*)

Set the logging level of this handler. level must be an int or a str.

setStream(*stream*)

Sets the StreamHandler's stream to the specified value, if it is different.

Returns the old stream, if the stream was changed, or None if it wasn't.

set_name(*name*)

terminator = '\n'

AFL.automation.APIServer.APIServer.Flask

class AFL.automation.APIServer.APIServer.**Flask**(*import_name: str*, *static_url_path: str | None = None*, *static_folder: str | PathLike | None = 'static'*, *static_host: str | None = None*, *host_matching: bool = False*, *subdomain_matching: bool = False*, *template_folder: str | PathLike | None = 'templates'*, *instance_path: str | None = None*, *instance_relative_config: bool = False*, *root_path: str | None = None*)

Bases: `Scaffold`

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see *open_resource()*.

Usually you create a *Flask* instance in your main module or in the `__init__.py` file of your package like this:

```
from flask import Flask
app = Flask(__name__)
```

About the First Parameter

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

Added in version 0.7: The *static_url_path*, *static_folder*, and *template_folder* parameters were added.

Added in version 0.8: The *instance_path* and *instance_relative_config* parameters were added.

Added in version 0.11: The *root_path* parameter was added.

Added in version 1.0: The `host_matching` and `static_host` parameters were added.

Added in version 1.0: The `subdomain_matching` parameter was added. Subdomain matching needs to be enabled manually now. Setting `SERVER_NAME` does not implicitly enable it.

Parameters

- **import_name** – the name of the application package
- **static_url_path** – can be used to specify a different path for the static files on the web. Defaults to the name of the *static_folder* folder.
- **static_folder** – The folder with static files that is served at `static_url_path`. Relative to the application `root_path` or an absolute path. Defaults to `'static'`.
- **static_host** – the host to use when adding the static route. Defaults to None. Required when using `host_matching=True` with a `static_folder` configured.
- **host_matching** – set `url_map.host_matching` attribute. Defaults to False.
- **subdomain_matching** – consider the subdomain relative to `SERVER_NAME` when matching routes. Defaults to False.
- **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
- **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.
- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** – The path to the root of the application files. This should only be set manually when it can't be detected automatically, such as for namespace packages.

__init__(*import_name: str*, *static_url_path: str | None = None*, *static_folder: str | PathLike | None = 'static'*, *static_host: str | None = None*, *host_matching: bool = False*, *subdomain_matching: bool = False*, *template_folder: str | PathLike | None = 'templates'*, *instance_path: str | None = None*, *instance_relative_config: bool = False*, *root_path: str | None = None*)

Methods

__init__(import_name[, static_url_path, ...])	
add_template_filter(f[, name])	Register a custom template filter.
add_template_global(f[, name])	Register a custom template global function.
add_template_test(f[, name])	Register a custom template test.
add_url_rule(rule[, endpoint, view_func, ...])	Register a rule for routing incoming requests and building URLs.
after_request(f)	Register a function to run after each request to this object.
app_context()	Create an `AppContext`.
async_to_sync(func)	Return a sync function that will run the coroutine function.
auto_find_instance_path()	Tries to locate the instance path if it was not provided to the constructor of the application class.
before_first_request(f)	Registers a function to be run before the first request to this instance of the application.
before_request(f)	Register a function to run before each request.
context_processor(f)	Registers a template context processor function.

continues on next page

Table 2 – continued from previous page

create_global_jinja_loader()	Creates the loader for the Jinja2 environment.
create_jinja_environment()	Create the Jinja environment based on *jinja_options* and the various Jinja-related methods of the app.
create_url_adapter(request)	Creates a URL adapter for the given request.
delete(rule, **options)	Shortcut for *route()* with `methods=["DELETE"]`.
dispatch_request()	Does the request dispatching.
do_teardown_appcontext([exc])	Called right before the application context is popped.
do_teardown_request([exc])	Called after the request is dispatched and the response is returned, right before the request context is popped.
endpoint(endpoint)	Decorate a view function to register it for the given endpoint.
ensure_sync(func)	Ensure that the function is synchronous for WSGI workers.
errorhandler(code_or_exception)	Register a function to handle errors by code or exception class.
finalize_request(rv[, from_error_handler])	Given the return value from a view function this finalizes the request by converting it into a response and invoking the postprocessing functions.
full_dispatch_request()	Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.
get(rule, **options)	Shortcut for *route()* with `methods=["GET"]`.
get_send_file_max_age(filename)	Used by *send_file()* to determine the `max_age` cache value for a given file path if it wasn't passed.
handle_exception(e)	Handle an exception that did not have an error handler associated with it, or that was raised from an error handler.
handle_http_exception(e)	Handles an HTTP exception.
handle_url_build_error(error, endpoint, values)	Called by *url_for()* if a `BuildError` was raised.
handle_user_exception(e)	This method is called whenever an exception occurs that should be handled.
inject_url_defaults(endpoint, values)	Injects the URL defaults for the given endpoint directly into the values dictionary passed.
iter_blueprints()	Iterates over all blueprints by the order they were registered.
log_exception(exc_info)	Logs an exception.
make_aborter()	Create the object to assign to *aborter*.
make_config([instance_relative])	Used to create the config attribute by the Flask constructor.
make_default_options_response()	This method is called to create the default `OPTIONS` response.
make_response(rv)	Convert the return value from a view function to an instance of *response_class*.
make_shell_context()	Returns the shell context for an interactive shell for this application.
open_instance_resource(resource[, mode])	Opens a resource from the application's instance folder (*instance_path*).
open_resource(resource[, mode])	Open a resource file relative to *root_path* for reading.
patch(rule, **options)	Shortcut for *route()* with `methods=["PATCH"]`.

continues on next page

Table 2 – continued from previous page

post(rule, **options)	Shortcut for route() with methods=["POST"].
preprocess_request()	Called before the request is dispatched.
process_response(response)	Can be overridden in order to modify the response object before it's sent to the WSGI server.
put(rule, **options)	Shortcut for route() with methods=["PUT"].
raise_routing_exception(request)	Intercept routing exceptions and possibly do something else.
redirect(location[, code])	Create a redirect response object.
register_blueprint(blueprint, **options)	Register a Blueprint on the application.
register_error_handler(code_or_exception, f)	Alternative error attach function to the errorhandler() decorator that is more straightforward to use for non decorator usage.
request_context(environ)	Create a RequestContext representing a WSGI environment.
route(rule, **options)	Decorate a view function to register it with the given URL rule and options.
run([host, port, debug, load_dotenv])	Runs the application on a local development server.
select_jinja_autoescape(filename)	Returns True if autoescaping should be active for the given template name.
send_static_file(filename)	The view function used to serve files from static_folder.
shell_context_processor(f)	Registers a shell context processor function.
should_ignore_error(error)	This is called to figure out if an error should be ignored or not as far as the teardown system is concerned.
teardown_appcontext(f)	Registers a function to be called when the application context is popped.
teardown_request(f)	Register a function to be called when the request context is popped.
template_filter([name])	A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example::.
template_global([name])	A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example::.
template_test([name])	A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example::.
test_cli_runner(**kwargs)	Create a CLI runner for testing CLI commands.
test_client([use_cookies])	Creates a test client for this application.
test_request_context(*args, **kwargs)	Create a RequestContext for a WSGI environment created from the given values.
trap_http_exception(e)	Checks if an HTTP exception should be trapped or not.
update_template_context(context)	Update the template context with some commonly used variables.
url_defaults(f)	Callback function for URL defaults for all view functions of the application.
url_for(endpoint, *[, _anchor, _method, ...])	Generate a URL to the given endpoint with the given values.

continues on next page

Table 2 – continued from previous page

url_value_preprocessor(f)	Register a URL value preprocessor function for all view functions in the application.
wsgi_app(environ, start_response)	The actual WSGI application. This is not implemented in *__call__()* so that middlewares can be applied without losing a reference to the app object. Instead of doing this::.

Attributes

debug	Whether debug mode is enabled.
default_config	Default configuration parameters.
env	What environment the app is running in.
got_first_request	This attribute is set to `True` if the application started handling the first request.
has_static_folder	`True` if *static_folder* is set.
jinja_env	The Jinja environment used to load templates.
jinja_loader	The Jinja loader for this object's templates.
jinja_options	Options that are passed to the Jinja environment in *create_jinja_environment()*.
json_decoder	The JSON decoder class to use.
json_encoder	The JSON encoder class to use.
logger	A standard Python `Logger` for the app, with the same name as *name*.
name	The name of the application.
permanent_session_lifetime	A `timedelta` which is used to set the expiration date of a permanent session.
propagate_exceptions	Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned.
secret_key	If a secret key is set, cryptographic components can use this to sign cookies and other things.
send_file_max_age_default	The default value for `max_age` for `send_file()`.
session_cookie_name	The name of the cookie set by the session interface.
session_interface	the session interface to use.
static_folder	The absolute path to the configured static folder.
static_url_path	The URL prefix that the static route will be accessible from.
templates_auto_reload	Reload templates when they are changed.
test_cli_runner_class	The `CliRunner` subclass, by default `FlaskCliRunner` that is used by *test_cli_runner()*.
test_client_class	The *test_client()* method creates an instance of this test client class.
testing	The testing flag.
use_x_sendfile	Enable this to use the `X-Sendfile` feature, assuming the server supports it, from `send_file()`.
instance_path	Holds the path to the instance folder.
config	The configuration dictionary as `Config`.
aborter	An instance of *aborter_class* created by *make_aborter()*.
json	Provides access to JSON methods.

continues on next page

Table 3 – continued from previous page

url_build_error_handlers	A list of functions that are called by *handle_url_build_error()* when *url_for()* raises a BuildError.
before_first_request_funcs	A list of functions that will be called at the beginning of the first request to this instance.
teardown_appcontext_funcs	A list of functions that are called when the application context is destroyed.
shell_context_processors	A list of shell context processor functions that should be run when a shell context is created.
blueprints	Maps registered blueprint names to blueprint objects.
extensions	a place where extensions can store application specific state.
url_map	The Map for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example::.
import_name	The name of the package or module that this object belongs to.
template_folder	The path to the templates folder, relative to *root_path*, to add to the template loader.
root_path	Absolute path to the package on the filesystem.
cli	The Click command group for registering CLI commands for this object.
view_functions	A dictionary mapping endpoint names to view functions.
error_handler_spec	A data structure of registered error handlers, in the format {scope: {code: {class: handler}}}.
before_request_funcs	A data structure of functions to call at the beginning of each request, in the format {scope: [functions]}.
after_request_funcs	A data structure of functions to call at the end of each request, in the format {scope: [functions]}.
teardown_request_funcs	A data structure of functions to call at the end of each request even if an exception is raised, in the format {scope: [functions]}.
template_context_processors	A data structure of functions to call to pass extra context values when rendering templates, in the format {scope: [functions]}.
url_value_preprocessors	A data structure of functions to call to modify the keyword arguments passed to the view function, in the format {scope: [functions]}.
url_default_functions	A data structure of functions to call to modify the keyword arguments when generating URLs, in the format {scope: [functions]}.

request_class

alias of Request

response_class

alias of Response

aborter_class

alias of Aborter

jinja_environment

alias of Environment

app_ctx_globals_class

alias of _AppCtxGlobals

config_class

alias of Config

testing

The testing flag. Set this to True to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate test helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE_EXCEPTIONS is not changed from the default it's implicitly enabled.

This attribute can also be configured from the config with the TESTING configuration key. Defaults to False.

secret_key

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the SECRET_KEY configuration key. Defaults to None.

property session_cookie_name: str

The name of the cookie set by the session interface.

Deprecated since version 2.2: Will be removed in Flask 2.3. Use app.config["SESSION_COOKIE_NAME"] instead.

permanent_session_lifetime

A timedelta which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the PERMANENT_SESSION_LIFETIME configuration key. Defaults to timedelta(days=31)

property send_file_max_age_default: timedelta | None

The default value for max_age for send_file(). The default is None, which tells the browser to use conditional requests instead of a timed cache.

Deprecated since version 2.2: Will be removed in Flask 2.3. Use app.config["SEND_FILE_MAX_AGE_DEFAULT"] instead.

Changed in version 2.0: Defaults to None instead of 12 hours.

property use_x_sendfile: bool

Enable this to use the X-Sendfile feature, assuming the server supports it, from send_file().

Deprecated since version 2.2: Will be removed in Flask 2.3. Use app.config["USE_X_SENDFILE"] instead.

property json_encoder: Type[JSONEncoder]

The JSON encoder class to use. Defaults to JSONEncoder.

Deprecated since version 2.2: Will be removed in Flask 2.3. Customize *json_provider_class* instead.

Added in version 0.10.

property json_decoder: Type[JSONDecoder]

The JSON decoder class to use. Defaults to `JSONDecoder`.

Deprecated since version 2.2: Will be removed in Flask 2.3. Customize *json_provider_class* instead.

Added in version 0.10.

json_provider_class

alias of `DefaultJSONProvider`

jinja_options: dict = {}

Options that are passed to the Jinja environment in *create_jinja_environment()*. Changing these options after the environment is created (accessing *jinja_env*) will have no effect.

Changed in version 1.1.0: This is a `dict` instead of an `ImmutableDict` to allow easier configuration.

default_config = {'APPLICATION_ROOT': '/', 'DEBUG': None, 'ENV': None, 'EXPLAIN_TEMPLATE_LOADING': False, 'JSONIFY_MIMETYPE': None, 'JSONIFY_PRETTYPRINT_REGULAR': None, 'JSON_AS_ASCII': None, 'JSON_SORT_KEYS': None, 'MAX_CONTENT_LENGTH': None, 'MAX_COOKIE_SIZE': 4093, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31), 'PREFERRED_URL_SCHEME': 'http', 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None, 'SEND_FILE_MAX_AGE_DEFAULT': None, 'SERVER_NAME': None, 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_SAMESITE': None, 'SESSION_COOKIE_SECURE': False, 'SESSION_REFRESH_EACH_REQUEST': True, 'TEMPLATES_AUTO_RELOAD': None, 'TESTING': False, 'TRAP_BAD_REQUEST_ERRORS': None, 'TRAP_HTTP_EXCEPTIONS': False, 'USE_X_SENDFILE': False}

Default configuration parameters.

url_rule_class

alias of `Rule`

url_map_class

alias of `Map`

test_client_class: Type[FlaskClient] | None = None

The *test_client()* method creates an instance of this test client class. Defaults to `FlaskClient`.

Added in version 0.7.

test_cli_runner_class: Type[FlaskCliRunner] | None = None

The `CliRunner` subclass, by default `FlaskCliRunner` that is used by *test_cli_runner()*. Its `__init__` method should take a Flask app object as the first argument.

Added in version 1.0.

session_interface: SessionInterface = <flask.sessions.SecureCookieSessionInterface object>

the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here.

Added in version 0.8.

__init__(*import_name: str*, *static_url_path: str | None = None*, *static_folder: str | PathLike | None = 'static'*, *static_host: str | None = None*, *host_matching: bool = False*, *subdomain_matching: bool = False*, *template_folder: str | PathLike | None = 'templates'*, *instance_path: str | None = None*, *instance_relative_config: bool = False*, *root_path: str | None = None*)

instance_path

Holds the path to the instance folder.

Added in version 0.8.

config

The configuration dictionary as `Config`. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

aborter

An instance of *aborter_class* created by *make_aborter()*. This is called by `flask.abort()` to raise HTTP errors, and can be called directly as well.

Added in version 2.2: Moved from `flask.abort`, which calls this object.

json: JSONProvider

Provides access to JSON methods. Functions in `flask.json` will call methods on this provider when the application context is active. Used for handling JSON requests and responses.

An instance of *json_provider_class*. Can be customized by changing that attribute on a subclass, or by assigning to this attribute afterwards.

The default, `DefaultJSONProvider`, uses Python's built-in `json` library. A different provider can use a different JSON library.

Added in version 2.2.

url_build_error_handlers: t.List[t.Callable[[Exception, str, t.Dict[str, t.Any]], str]]

A list of functions that are called by *handle_url_build_error()* when *url_for()* raises a `BuildError`. Each function is called with `error`, `endpoint` and `values`. If a function returns `None` or raises a `BuildError`, it is skipped. Otherwise, its return value is returned by `url_for`.

Added in version 0.9.

before_first_request_funcs: t.List[ft.BeforeFirstRequestCallable]

A list of functions that will be called at the beginning of the first request to this instance. To register a function, use the *before_first_request()* decorator.

Deprecated since version 2.2: Will be removed in Flask 2.3. Run setup code when creating the application instead.

Added in version 0.8.

teardown_appcontext_funcs: t.List[ft.TeardownCallable]

A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

Added in version 0.9.

shell_context_processors: t.List[ft.ShellContextProcessorCallable]

A list of shell context processor functions that should be run when a shell context is created.

Added in version 0.11.

blueprints: t.Dict[str, 'Blueprint']

Maps registered blueprint names to blueprint objects. The dict retains the order the blueprints were registered in. Blueprints can be registered multiple times, this dict does not track how often they were attached.

Added in version 0.7.

extensions: dict

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things.

The key must match the name of the extension module. For example in case of a "Flask-Foo" extension in *flask_foo*, the key would be `'foo'`.

Added in version 0.7.

url_map

The `Map` for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
        return ','.join(super(ListConverter, self).to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

property name: str

The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

Added in version 0.8.

property propagate_exceptions: bool

Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned.

Deprecated since version 2.2: Will be removed in Flask 2.3.

Added in version 0.7.

property logger: Logger

A standard Python `Logger` for the app, with the same name as *name*.

In debug mode, the logger's `level` will be set to `DEBUG`.

If there are no handlers configured, a default handler will be added. See /logging for more information.

Changed in version 1.1.0: The logger takes the same name as *name* rather than hard-coding `"flask.app"`.

Changed in version 1.0.0: Behavior was simplified. The logger is always named `"flask.app"`. The level is only set during configuration, it doesn't check `app.debug` each time. Only one format is used, not different ones depending on `app.debug`. No handlers are removed, and a handler is only added if no handlers are already configured.

Added in version 0.3.

property jinja_env: Environment

The Jinja environment used to load templates.

The environment is created the first time this property is accessed. Changing *jinja_options* after that will have no effect.

property got_first_request: bool

This attribute is set to True if the application started handling the first request.

Added in version 0.8.

make_config(*instance_relative: bool = False*) → Config

Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

Added in version 0.8.

make_aborter() → Aborter

Create the object to assign to *aborter*. That object is called by flask.abort() to raise HTTP errors, and can be called directly as well.

By default, this creates an instance of *aborter_class*, which defaults to werkzeug.exceptions.Aborter.

Added in version 2.2.

auto_find_instance_path() → str

Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named instance next to your main file or the package.

Added in version 0.8.

open_instance_resource(*resource: str*, *mode: str = 'rb'*) → IO

Opens a resource from the application's instance folder (*instance_path*). Otherwise works like *open_resource()*. Instance resources can also be opened for writing.

Parameters

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

property templates_auto_reload: bool

Reload templates when they are changed. Used by *create_jinja_environment()*. It is enabled by default in debug mode.

Deprecated since version 2.2: Will be removed in Flask 2.3. Use app.config["TEMPLATES_AUTO_RELOAD"] instead.

Added in version 1.0: This property was added but the underlying config and behavior already existed.

create_jinja_environment() → Environment

Create the Jinja environment based on *jinja_options* and the various Jinja-related methods of the app. Changing *jinja_options* after this will have no effect. Also adds Flask-related globals and filters to the environment.

Changed in version 0.11: Environment.auto_reload set in accordance with TEMPLATES_AUTO_RELOAD configuration option.

Added in version 0.5.

create_global_jinja_loader() → DispatchingJinjaLoader

Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the *jinja_loader()* function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

Added in version 0.7.

select_jinja_autoescape(*filename: str*) → bool

Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

Changed in version 2.2: Autoescaping is now enabled by default for `.svg` files.

Added in version 0.5.

update_template_context(*context: dict*) → None

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

Parameters
context – the context as a dictionary that is updated in place to add extra variables.

make_shell_context() → dict

Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

Added in version 0.11.

property env: str

What environment the app is running in. This maps to the `ENV` config key.

Do not enable development when deploying in production.

Default: `'production'`

Deprecated since version 2.2: Will be removed in Flask 2.3.

property debug: bool

Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. This maps to the `DEBUG` config key. It may not behave as expected if set late.

Do not enable debug mode when deploying in production.

Default: `False`

run(*host: str | None = None*, *port: int | None = None*, *debug: bool | None = None*, *load_dotenv: bool = True*, ***options: Any*) → None

Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see /deploying/index for WSGI server recommendations.

If the *debug* flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's `run` support.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke *run()* with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

Parameters

- **host** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'` or the host in the `SERVER_NAME` config variable if present.
- **port** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **debug** – if given, enable or disable debug mode. See *debug*.
- **load_dotenv** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.
- **options** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from `.env` and `.flaskenv` files.

The `FLASK_DEBUG` environment variable will override *debug*.

Threaded mode is enabled by default.

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

test_client(*use_cookies: bool = True*, ***kwargs: Any*) → FlaskClient

Creates a test client for this application. For information about unit testing head over to /testing.

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the *testing* attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's *test_client_class* constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

Added in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the *test_client_class* attribute.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of *test_client_class*.

test_cli_runner(***kwargs: Any*) → FlaskCliRunner

Create a CLI runner for testing CLI commands. See testing-cli.

Returns an instance of *test_cli_runner_class*, by default `FlaskCliRunner`. The Flask app object is passed as the first argument.

Added in version 1.0.

register_blueprint(*blueprint: Blueprint*, ***options: Any*) → None

Register a `Blueprint` on the application. Keyword arguments passed to this method will override the defaults set on the blueprint.

Calls the blueprint's `register()` method after recording the blueprint in the application's *blueprints*.

Parameters

- **blueprint** – The blueprint to register.
- **url_prefix** – Blueprint routes will be prefixed with this.
- **subdomain** – Blueprint routes will match on this subdomain.
- **url_defaults** – Blueprint routes will use these default values for view arguments.
- **options** – Additional keyword arguments are passed to `BlueprintSetupState`. They can be accessed in `record()` callbacks.

Changed in version 2.0.1: The `name` option can be used to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for `url_for`.

Added in version 0.7.

iter_blueprints() → ValuesView[Blueprint]

Iterates over all blueprints by the order they were registered.

Added in version 0.11.

add_url_rule(*rule: str*, *endpoint: str | None = None*, *view_func: Callable[[...], Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | WSGIApplication] | Callable[[...], Awaitable[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | WSGIApplication]] | None = None*, *provide_automatic_options: bool | None = None*, ***options: Any*) → None

Register a rule for routing incoming requests and building URLs. The *route()* decorator is a shortcut to call this with the `view_func` argument. These are equivalent:

```
@app.route("/")
def index():
    ...
```

```
def index():
    ...

app.add_url_rule("/", view_func=index)
```

See url-route-registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed. An error will be raised if a function has already been registered for the endpoint.

The `methods` parameter defaults to `["GET"]`. `HEAD` is always added automatically, and `OPTIONS` is added automatically by default.

`view_func` does not necessarily need to be passed, but if the rule should participate in routing an endpoint name must be associated with a view function at some point with the *endpoint()* decorator.

```
app.add_url_rule("/", endpoint="index")

@app.endpoint("index")
def index():
    ...
```

If `view_func` has a `required_methods` attribute, those methods are added to the passed and automatic methods. If it has a `provide_automatic_methods` attribute, it is used as the default if the parameter is not passed.

Parameters

- **rule** – The URL rule string.
- **endpoint** – The endpoint name to associate with the rule and view function. Used when routing and building URLs. Defaults to `view_func.__name__`.
- **view_func** – The view function to associate with the endpoint name.

- **provide_automatic_options** – Add the OPTIONS method and respond to OPTIONS requests automatically.
- **options** – Extra options passed to the Rule object.

template_filter(*name: str | None = None*) → Callable[[T_template_filter], T_template_filter]

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

Parameters
name – the optional name of the filter, otherwise the function name will be used.

add_template_filter(*f: Callable[[...], Any]*, *name: str | None = None*) → None

Register a custom template filter. Works exactly like the *template_filter()* decorator.

Parameters
name – the optional name of the filter, otherwise the function name will be used.

template_test(*name: str | None = None*) → Callable[[T_template_test], T_template_test]

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

Added in version 0.10.

Parameters
name – the optional name of the test, otherwise the function name will be used.

add_template_test(*f: Callable[[...], bool]*, *name: str | None = None*) → None

Register a custom template test. Works exactly like the *template_test()* decorator.

Added in version 0.10.

Parameters
name – the optional name of the test, otherwise the function name will be used.

template_global(*name: str | None = None*) → Callable[[T_template_global], T_template_global]

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

Added in version 0.10.

Parameters
name – the optional name of the global function, otherwise the function name will be used.

add_template_global(*f: Callable[[...], Any]*, *name: str | None = None*) → None

Register a custom template global function. Works exactly like the *template_global()* decorator.

Added in version 0.10.

Parameters
name – the optional name of the global function, otherwise the function name will be used.

before_first_request(*f: T_before_first_request*) → T_before_first_request

Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

Deprecated since version 2.2: Will be removed in Flask 2.3. Run setup code when creating the application instead.

Added in version 0.8.

teardown_appcontext(*f: T_teardown*) → T_teardown

Registers a function to be called when the application context is popped. The application context is typically popped after the request context for each request, at the end of CLI commands, or after a manually pushed context ends.

```
with app.app_context():
    ...
```

When the `with` block exits (or `ctx.pop()` is called), the teardown functions are called just before the app context is made inactive. Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an unhandled exception it will be passed an error object. If an *errorhandler()* is registered, it will handle the exception and the teardown will not receive it.

Teardown functions must avoid raising exceptions. If they execute code that might fail they must surround that code with a `try/except` block and log any errors.

The return values of teardown functions are ignored.

Added in version 0.9.

shell_context_processor(*f: T_shell_context_processor*) → T_shell_context_processor

Registers a shell context processor function.

Added in version 0.11.

handle_http_exception(*e: HTTPException*) → HTTPException | Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

Changed in version 1.0.3: `RoutingException`, used internally for actions such as slash redirects during routing, is not passed to error handlers.

Changed in version 1.0: Exceptions are looked up by code *and* by MRO, so `HTTPException` subclasses can be handled with a catch-all handler for the base `HTTPException`.

Added in version 0.3.

trap_http_exception(*e: Exception*) → bool

Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

Changed in version 1.0: Bad request errors are not trapped by default in debug mode.

Added in version 0.8.

handle_user_exception(*e: Exception*) → HTTPException | Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication

This method is called whenever an exception occurs that should be handled. A special case is `HTTPException` which is forwarded to the *handle_http_exception()* method. This function will either return a response value or reraise the exception with the same traceback.

Changed in version 1.0: Key errors raised from request data like `form` show the bad key in debug mode rather than a generic bad request message.

Added in version 0.7.

handle_exception(*e: Exception*) → Response

Handle an exception that did not have an error handler associated with it, or that was raised from an error handler. This always causes a 500 `InternalServerError`.

Always sends the `got_request_exception` signal.

If *propagate_exceptions* is `True`, such as in debug mode, the error will be re-raised so that the debugger can display it. Otherwise, the original exception is logged, and an `InternalServerError` is returned.

If an error handler is registered for `InternalServerError` or `500`, it will be used. For consistency, the handler will always receive the `InternalServerError`. The original unhandled exception is available as `e.original_exception`.

Changed in version 1.1.0: Always passes the `InternalServerError` instance to the handler, setting `original_exception` to the unhandled error.

Changed in version 1.1.0: `after_request` functions and other finalization is done even for the default 500 response when there is no handler.

Added in version 0.3.

log_exception(*exc_info: Tuple[type, BaseException, TracebackType] | Tuple[None, None, None]*) → None

Logs an exception. This is called by *handle_exception()* if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the *logger*.

Added in version 0.8.

dispatch_request() → Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication

Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call *make_response()*.

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new *full_dispatch_request()*.

full_dispatch_request() → Response

Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

Added in version 0.7.

finalize_request(*rv: Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | WSGIApplication | HTTPException*, *from_error_handler: bool = False*) → Response

Given the return value from a view function this finalizes the request by converting it into a response and invoking the postprocessing functions. This is invoked for both normal request dispatching as well as error handlers.

Because this means that it might be called as a result of a failure a special safe mode is available which can be enabled with the *from_error_handler* flag. If enabled, failures in response processing will be logged and otherwise ignored.

Internal

make_default_options_response() → Response

This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.

Added in version 0.7.

should_ignore_error(*error: BaseException | None*) → bool

This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

Added in version 0.10.

ensure_sync(*func: Callable*) → Callable

Ensure that the function is synchronous for WSGI workers. Plain `def` functions are returned as-is. `async def` functions are wrapped to run and wait for the response.

Override this method to change how the app runs async views.

Added in version 2.0.

async_to_sync(*func: Callable[[...], Coroutine]*) → Callable[[...], Any]

Return a sync function that will run the coroutine function.

```
result = app.async_to_sync(func)(*args, **kwargs)
```

Override this method to change how the app converts async code to be synchronously callable.

Added in version 2.0.

url_for(*endpoint: str*, *, *_anchor: str | None = None*, *_method: str | None = None*, *_scheme: str | None = None*, *_external: bool | None = None*, ***values: Any*) → str

Generate a URL to the given endpoint with the given values.

This is called by `flask.url_for()`, and can be called directly as well.

An *endpoint* is the name of a URL rule, usually added with *@app.route()*, and usually the same name as the view function. A route defined in a `Blueprint` will prepend the blueprint's name separated by a `.` to the endpoint.

In some cases, such as email messages, you want URLs to include the scheme and domain, like `https://example.com/hello`. When not in an active request, URLs will be external by default, but this requires setting `SERVER_NAME` so Flask knows what domain to use. `APPLICATION_ROOT` and `PREFERRED_URL_SCHEME` should also be configured as needed. This config is only used when not in an active request.

Functions can be decorated with *url_defaults()* to modify keyword arguments before the URL is built.

If building fails for some reason, such as an unknown endpoint or incorrect values, the app's *handle_url_build_error()* method is called. If that returns a string, that is returned, otherwise a `BuildError` is raised.

Parameters

- **endpoint** – The endpoint name associated with the URL to generate. If this starts with a `.`, the current blueprint name (if any) will be used.
- **_anchor** – If given, append this as `#anchor` to the URL.
- **_method** – If given, generate the URL associated with this method for the endpoint.
- **_scheme** – If given, the URL will have this scheme if it is external.
- **_external** – If given, prefer the URL to be internal (False) or require it to be external (True). External URLs include the scheme and domain. When not in an active request, URLs are external by default.
- **values** – Values to use for the variable parts of the URL rule. Unknown keys are appended as query string arguments, like `?a=b&c=d`.

Added in version 2.2: Moved from `flask.url_for`, which calls this method.

redirect(*location: str*, *code: int = 302*) → Response

Create a redirect response object.

This is called by `flask.redirect()`, and can be called directly as well.

Parameters

- **location** – The URL to redirect to.
- **code** – The status code for the redirect.

Added in version 2.2: Moved from `flask.redirect`, which calls this method.

make_response(*rv: Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication*) → Response

Convert the return value from a view function to an instance of *response_class*.

Parameters

rv – the return value from the view function. The view function must return a response. Returning `None`, or the view ending without returning, is not allowed. The following types are allowed for `view_rv`:

str
A response object is created with the string encoded to UTF-8 as the body.

bytes
A response object is created with the bytes as the body.

dict
A dictionary that will be jsonify'd before being returned.

list
A list that will be jsonify'd before being returned.

generator or iterator
A generator that returns `str` or `bytes` to be streamed as the response.

tuple
Either `(body, status, headers)`, `(body, status)`, or `(body, headers)`, where `body` is any of the other types allowed here, `status` is a string or an integer, and `headers` is a dictionary or a list of `(key, value)` tuples. If `body` is a *response_class* instance, `status` overwrites the exiting value and `headers` are extended.

response_class
The object is returned unchanged.

other Response class
The object is coerced to *response_class*.

callable()
The function is called as a WSGI application. The result is used to create a response object.

Changed in version 2.2: A generator will be converted to a streaming response. A list will be converted to a JSON response.

Changed in version 1.1: A dict will be converted to a JSON response.

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

create_url_adapter(*request: Request | None*) → MapAdapter | None

Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.

Added in version 0.6.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

Changed in version 1.0: `SERVER_NAME` no longer implicitly enables subdomain matching. Use `subdomain_matching` instead.

after_request(*f: T_after_request*) → T_after_request

Register a function to run after each request to this object.

The function is called with the response object, and must return a response object. This allows the functions to modify or replace the response before it is sent.

If a function raises an exception, any remaining `after_request` functions will not be called. Therefore, this should not be used for actions that must execute, such as to close resources. Use *teardown_request()* for that.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use `Blueprint.after_app_request()`.

before_request(*f: T_before_request*) → T_before_request

Register a function to run before each request.

For example, this can be used to open a database connection, or to load the logged in user from the session.

```
@app.before_request
def load_user():
    if "user_id" in session:
        g.user = db.session.get(session["user_id"])
```

The function will be called without any arguments. If it returns a non-`None` value, the value is handled as if it was the return value from the view, and further request handling is stopped.

This is available on both app and blueprint objects. When used on an app, this executes before every request. When used on a blueprint, this executes before every request that the blueprint handles. To register with a blueprint and execute before every request, use `Blueprint.before_app_request()`.

context_processor(*f: T_template_context_processor*) → T_template_context_processor

Registers a template context processor function. These functions run before rendering a template. The keys of the returned dict are added as variables available in the template.

This is available on both app and blueprint objects. When used on an app, this is called for every rendered template. When used on a blueprint, this is called for templates rendered from the blueprint's views. To register with a blueprint and affect every template, use `Blueprint.app_context_processor()`.

delete(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Shortcut for *route()* with `methods=["DELETE"]`.

Added in version 2.0.

endpoint(*endpoint: str*) → Callable[[F], F]

Decorate a view function to register it for the given endpoint. Used if a rule is added without a `view_func` with *add_url_rule()*.

```
app.add_url_rule("/ex", endpoint="example")

@app.endpoint("example")
```

(continues on next page)

(continued from previous page)

```
def example():
    ...
```

Parameters
endpoint – The endpoint name to associate with the view function.

errorhandler(*code_or_exception: Type[Exception] | int*) → Callable[[T_error_handler], T_error_handler]

Register a function to handle errors by code or exception class.

A decorator that is used to register a function given an error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

This is available on both app and blueprint objects. When used on an app, this can handle errors from every request. When used on a blueprint, this can handle errors from requests that the blueprint handles. To register with a blueprint and affect every request, use `Blueprint.app_errorhandler()`.

Added in version 0.7: Use *register_error_handler()* instead of modifying *error_handler_spec* directly, for application wide error handlers.

Added in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters
code_or_exception – the code as integer for the handler, or an arbitrary exception

get(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Shortcut for *route()* with `methods=["GET"]`.

Added in version 2.0.

get_send_file_max_age(*filename: str | None*) → int | None

Used by *send_file()* to determine the `max_age` cache value for a given file path if it wasn't passed.

By default, this returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`. This defaults to `None`, which tells the browser to use conditional requests instead of a timed cache, which is usually preferable.

Changed in version 2.0: The default configuration is `None` instead of 12 hours.

Added in version 0.9.

property has_static_folder: bool

`True` if *static_folder* is set.

Added in version 0.5.

inject_url_defaults(*endpoint: str*, *values: dict*) → None

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

Added in version 0.7.

property jinja_loader: FileSystemLoader | None

The Jinja loader for this object's templates. By default this is a class `jinja2.loaders.FileSystemLoader` to *template_folder* if it is set.

Added in version 0.5.

open_resource(*resource: str*, *mode: str = 'rb'*) → IO

Open a resource file relative to *root_path* for reading.

For example, if the file `schema.sql` is next to the file `app.py` where the `Flask` app is defined, it can be opened with:

```
with app.open_resource("schema.sql") as f:
    conn.executescript(f.read())
```

Parameters

- **resource** – Path to the resource relative to *root_path*.
- **mode** – Open the file in this mode. Only reading is supported, valid values are "r" (or "rt") and "rb".

patch(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Shortcut for *route()* with `methods=["PATCH"]`.

Added in version 2.0.

post(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Shortcut for *route()* with `methods=["POST"]`.

Added in version 2.0.

put(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Shortcut for *route()* with `methods=["PUT"]`.

Added in version 2.0.

register_error_handler(*code_or_exception: Type[Exception] | int*, *f: Callable[[Any], Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication]*) → None

Alternative error attach function to the *errorhandler()* decorator that is more straightforward to use for non decorator usage.

Added in version 0.7.

route(*rule: str*, ***options: Any*) → Callable[[T_route], T_route]

Decorate a view function to register it with the given URL rule and options. Calls *add_url_rule()*, which has more details about the implementation.

```
@app.route("/")
def index():
    return "Hello, World!"
```

See url-route-registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed.

The `methods` parameter defaults to `["GET"]`. `HEAD` and `OPTIONS` are added automatically.

Parameters

- **rule** – The URL rule string.
- **options** – Extra options passed to the `Rule` object.

send_static_file(*filename: str*) → Response

The view function used to serve files from *static_folder*. A route is automatically registered for this view at *static_url_path* if *static_folder* is set.

Added in version 0.5.

property static_folder: str | None

The absolute path to the configured static folder. `None` if no static folder is set.

property static_url_path: str | None

The URL prefix that the static route will be accessible from.

If it was not configured during init, it is derived from *static_folder*.

teardown_request(*f: T_teardown*) → T_teardown

Register a function to be called when the request context is popped. Typically this happens at the end of each request, but contexts may be pushed manually as well during testing.

```
with app.test_request_context():
    ...
```

When the `with` block exits (or `ctx.pop()` is called), the teardown functions are called just before the request context is made inactive.

When a teardown function was called because of an unhandled exception it will be passed an error object. If an *errorhandler()* is registered, it will handle the exception and the teardown will not receive it.

Teardown functions must avoid raising exceptions. If they execute code that might fail they must surround that code with a `try/except` block and log any errors.

The return values of teardown functions are ignored.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use `Blueprint.teardown_app_request()`.

url_defaults(*f: T_url_defaults*) → T_url_defaults

Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect every request, use `Blueprint.app_url_defaults()`.

url_value_preprocessor(*f: T_url_value_preprocessor*) → T_url_value_preprocessor

Register a URL value preprocessor function for all view functions in the application. These functions will be called before the *before_request()* functions.

The function can modify the values captured from the matched url before they are passed to the view. For example, this can be used to pop a common language code value and place it in `g` rather than pass it to every view.

The function is passed the endpoint name and values dict. The return value is ignored.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect every request, use `Blueprint.app_url_value_preprocessor()`.

import_name

The name of the package or module that this object belongs to. Do not change this once it is set by the constructor.

template_folder

The path to the templates folder, relative to *root_path*, to add to the template loader. `None` if templates should not be added.

root_path

Absolute path to the package on the filesystem. Used to look up resources contained in the package.

cli

The Click command group for registering CLI commands for this object. The commands are available from the `flask` command once the application has been discovered and blueprints have been registered.

view_functions: t.Dict[str, t.Callable]

A dictionary mapping endpoint names to view functions.

To register a view function, use the *route()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

error_handler_spec: t.Dict[ft.AppOrBlueprintKey, t.Dict[t.Optional[int], t.Dict[t.Type[Exception], ft.ErrorHandlerCallable]]]

A data structure of registered error handlers, in the format `{scope: {code: {class: handler}}}`. The `scope` key is the name of a blueprint the handlers are active for, or `None` for all requests. The `code` key is the HTTP status code for `HTTPException`, or `None` for other exceptions. The innermost dictionary maps exception classes to handler functions.

To register an error handler, use the *errorhandler()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

before_request_funcs: t.Dict[ft.AppOrBlueprintKey, t.List[ft.BeforeRequestCallable]]

A data structure of functions to call at the beginning of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *before_request()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

after_request_funcs: t.Dict[ft.AppOrBlueprintKey, t.List[ft.AfterRequestCallable]]

A data structure of functions to call at the end of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *after_request()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

teardown_request_funcs: t.Dict[ft.AppOrBlueprintKey, t.List[ft.TeardownCallable]]

A data structure of functions to call at the end of each request even if an exception is raised, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *teardown_request()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

template_context_processors: t.Dict[ft.AppOrBlueprintKey, t.List[ft.TemplateContextProcessorCallable]]

A data structure of functions to call to pass extra context values when rendering templates, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *context_processor()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

url_value_preprocessors: t.Dict[ft.AppOrBlueprintKey, t.List[ft.URLValuePreprocessorCallable]]

A data structure of functions to call to modify the keyword arguments passed to the view function, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *url_value_preprocessor()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

url_default_functions: t.Dict[ft.AppOrBlueprintKey, t.List[ft.URLDefaultCallable]]

A data structure of functions to call to modify the keyword arguments when generating URLs, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the *url_defaults()* decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

handle_url_build_error(*error: BuildError*, *endpoint: str*, *values: Dict[str, Any]*) → str

Called by *url_for()* if a `BuildError` was raised. If this returns a value, it will be returned by `url_for`, otherwise the error will be re-raised.

Each function in *url_build_error_handlers* is called with `error`, `endpoint` and `values`. If a function returns `None` or raises a `BuildError`, it is skipped. Otherwise, its return value is returned by `url_for`.

Parameters

- **error** – The active `BuildError` being handled.
- **endpoint** – The endpoint being built.
- **values** – The keyword arguments passed to `url_for`.

preprocess_request() → Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int] | Tuple[Response | str | bytes | List[Any] | Mapping[str, Any] | Iterator[str] | Iterator[bytes], int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication | None

Called before the request is dispatched. Calls *url_value_preprocessors* registered with the app and the current blueprint (if any). Then calls *before_request_funcs* registered with the app and the blueprint.

If any *before_request()* handler returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

process_response(*response: Response*) → Response

Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the *after_request()* decorated functions.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters
response – a *response_class* object.

Returns
a new response object or the same, has to be an instance of *response_class*.

do_teardown_request(*exc: BaseException | None = <object object>*) → None

Called after the request is dispatched and the response is returned, right before the request context is popped.

This calls all functions decorated with *teardown_request()*, and `Blueprint.teardown_request()` if a blueprint handled the request. Finally, the `request_tearing_down` signal is sent.

This is called by `RequestContext.pop()`, which may be delayed during testing to maintain access to resources.

Parameters
exc – An unhandled exception raised while dispatching the request. Detected from the current exception information if not passed. Passed to each teardown function.

Changed in version 0.9: Added the `exc` argument.

do_teardown_appcontext(*exc: BaseException | None = <object object>*) → None

Called right before the application context is popped.

When handling a request, the application context is popped after the request context. See *do_teardown_request()*.

This calls all functions decorated with *teardown_appcontext()*. Then the `appcontext_tearing_down` signal is sent.

This is called by `AppContext.pop()`.

Added in version 0.9.

app_context() → AppContext

Create an `AppContext`. Use as a `with` block to push the context, which will make `current_app` point at this application.

An application context is automatically pushed by `RequestContext.push()` when handling a request, and when running a CLI command. Use this to manually create a context outside of these situations.

```
with app.app_context():
    init_db()
```

See /appcontext.

Added in version 0.9.

request_context(*environ: dict*) → RequestContext

Create a `RequestContext` representing a WSGI environment. Use a `with` block to push the context, which will make `request` point at this request.

See /reqcontext.

Typically you should not call this from your own code. A request context is automatically pushed by the *wsgi_app()* when handling a request. Use *test_request_context()* to create an environment and context instead of this method.

Parameters
environ – a WSGI environment

test_request_context(**args: Any*, ***kwargs: Any*) → RequestContext

Create a `RequestContext` for a WSGI environment created from the given values. This is mostly useful during testing, where you may want to run a function that uses request data without dispatching a full request.

See /reqcontext.

Use a `with` block to push the context, which will make `request` point at the request for the created environment.

```
with test_request_context(...):
    generate_report()
```

When using the shell, it may be easier to push and pop the context manually to avoid indentation.

```
ctx = app.test_request_context(...)
ctx.push()
...
ctx.pop()
```

Takes the same arguments as Werkzeug's `EnvironBuilder`, with some defaults from the application. See the linked Werkzeug docs for most of the available arguments. Flask-specific behavior is listed here.

Parameters

- **path** – URL path being requested.
- **base_url** – Base URL where the app is being served, which `path` is relative to. If not given, built from `PREFERRED_URL_SCHEME`, `subdomain`, `SERVER_NAME`, and `APPLICATION_ROOT`.
- **subdomain** – Subdomain name to append to `SERVER_NAME`.
- **url_scheme** – Scheme to use instead of `PREFERRED_URL_SCHEME`.
- **data** – The request body, either as a string or a dict of form keys and values.
- **json** – If given, this is serialized as JSON and passed as `data`. Also defaults `content_type` to `application/json`.
- **args** – other positional arguments passed to `EnvironBuilder`.

- **kwargs** – other keyword arguments passed to EnvironBuilder.

wsgi_app(*environ: dict*, *start_response: Callable*) → Any

The actual WSGI application. This is not implemented in *__call__()* so that middlewares can be applied without losing a reference to the app object. Instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

Changed in version 0.7: Teardown events for the request and app contexts are called even if an unhandled error occurs. Other events may not be called depending on when an error occurs during dispatch. See callbacks-and-errors.

Parameters

- **environ** – A WSGI environment.
- **start_response** – A callable accepting a status code, a list of headers, and an optional exception context to start the response.

__call__(*environ: dict*, *start_response: Callable*) → Any

The WSGI server calls the Flask application object as the WSGI application. This calls *wsgi_app()*, which can be wrapped to apply middleware.

AFL.automation.APIServer.APIServer.IPVersion

class AFL.automation.APIServer.APIServer.**IPVersion**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *Enum*

__init__(**args*, ***kwds*)

Attributes

V4Only
V6Only
All

V4Only = 1

V6Only = 2

All = 3

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

AFL.automation.APIServer.APIServer.JWTManager

class AFL.automation.APIServer.APIServer.**JWTManager**(*app:* Flask | *None* = *None*, *add_context_processor:* *bool* = *False*)

Bases: object

An object used to hold JWT settings and callback functions for the Flask-JWT-Extended extension.

Instances of *JWTManager* are *not* bound to specific apps, so you can create one in the main body of your code and then bind it to your app in a factory function.

__init__(*app:* Flask | *None* = *None*, *add_context_processor:* *bool* = *False*) → None

Create the JWTManager instance. You can either pass a flask application in directly here to register this extension with the flask app, or call init_app after creating this object (in a factory pattern).

Parameters

- **app** – The Flask Application object
- **add_context_processor** – Controls if *current_user* is should be added to flasks template context (and thus be available for use in Jinja templates). Defaults to `False`.

Methods

__init__([app, add_context_processor])	Create the JWTManager instance.
additional_claims_loader(callback)	This decorator sets the callback function used to add additional claims when creating a JWT.
additional_headers_loader(callback)	This decorator sets the callback function used to add additional headers when creating a JWT.
decode_key_loader(callback)	This decorator sets the callback function for dynamically setting the JWT decode key based on the **UNVERIFIED** contents of the token.
encode_key_loader(callback)	This decorator sets the callback function for dynamically setting the JWT encode key based on the tokens identity.
expired_token_loader(callback)	This decorator sets the callback function for returning a custom response when an expired JWT is encountered.
init_app(app[, add_context_processor])	Register this extension with the flask app.
invalid_token_loader(callback)	This decorator sets the callback function for returning a custom response when an invalid JWT is encountered.
needs_fresh_token_loader(callback)	This decorator sets the callback function for returning a custom response when a valid and non-fresh token is used on an endpoint that is marked as `fresh=True`.
revoked_token_loader(callback)	This decorator sets the callback function for returning a custom response when a revoked token is encountered.
token_in_blocklist_loader(callback)	This decorator sets the callback function used to check if a JWT has been revoked.
token_verification_failed_loader(callback)	This decorator sets the callback function used to return a custom response when the claims verification check fails.
token_verification_loader(callback)	This decorator sets the callback function used for custom verification of a valid JWT.
unauthorized_loader(callback)	This decorator sets the callback function used to return a custom response when no JWT is present.
user_identity_loader(callback)	This decorator sets the callback function used to convert an identity to a string when creating JWTs.
user_lookup_error_loader(callback)	This decorator sets the callback function used to return a custom response when loading a user via `user_lookup_loader()` fails.
user_lookup_loader(callback)	This decorator sets the callback function used to convert a JWT into a python object that can be used in a protected endpoint.

__init__(*app:* Flask | *None* = *None*, *add_context_processor:* *bool* = *False*) → None

Create the JWTManager instance. You can either pass a flask application in directly here to register this extension with the flask app, or call init_app after creating this object (in a factory pattern).

Parameters

- **app** – The Flask Application object
- **add_context_processor** – Controls if *current_user* is should be added to flasks template context (and thus be available for use in Jinja templates). Defaults to `False`.

init_app(*app:* Flask, *add_context_processor: bool = False*) → None

Register this extension with the flask app.

Parameters

- **app** – The Flask Application object
- **add_context_processor** – Controls if *current_user* is should be added to flasks template context (and thus be available for use in Jinja templates). Defaults to `False`.

additional_claims_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to add additional claims when creating a JWT. The claims returned by this function will be merged with any claims passed in via the `additional_claims` argument to `create_access_token()` or `create_refresh_token()`.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return a dictionary of claims to add to the JWT.

additional_headers_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to add additional headers when creating a JWT. The headers returned by this function will be merged with any headers passed in via the `additional_headers` argument to `create_access_token()` or `create_refresh_token()`.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return a dictionary of headers to add to the JWT.

decode_key_loader(*callback: Callable*) → Callable

This decorator sets the callback function for dynamically setting the JWT decode key based on the **UNVERIFIED** contents of the token. Think carefully before using this functionality, in most cases you probably don't need it.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the unverified JWT.

The second argument is a dictionary containing the payload data of the unverified JWT.

The decorated function must return a *string* that is used to decode and verify the token.

encode_key_loader(*callback: Callable*) → Callable

This decorator sets the callback function for dynamically setting the JWT encode key based on the tokens identity. Think carefully before using this functionality, in most cases you probably don't need it.

The decorated function must take **one** argument.

The argument is the identity used to create this JWT.

The decorated function must return a *string* which is the secrete key used to encode the JWT.

expired_token_loader(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when an expired JWT is encountered.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

invalid_token_loader(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when an invalid JWT is encountered.

This decorator sets the callback function that will be used if an invalid JWT attempts to access a protected endpoint.

The decorated function must take **one** argument.

The argument is a string which contains the reason why a token is invalid.

The decorated function must return a Flask Response.

needs_fresh_token_loader(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when a valid and non-fresh token is used on an endpoint that is marked as `fresh=True`.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

revoked_token_loader(*callback: Callable*) → Callable

This decorator sets the callback function for returning a custom response when a revoked token is encountered.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

token_in_blocklist_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to check if a JWT has been revoked.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must be return `True` if the token has been revoked, `False` otherwise.

token_verification_failed_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when the claims verification check fails.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

token_verification_loader(*callback: Callable*) → Callable

This decorator sets the callback function used for custom verification of a valid JWT.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return `True` if the token is valid, or `False` otherwise.

unauthorized_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when no JWT is present.

The decorated function must take **one** argument.

The argument is a string that explains why the JWT could not be found.

The decorated function must return a Flask Response.

user_identity_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to convert an identity to a string when creating JWTs. This is useful for using objects (such as SQLAlchemy instances) as the identity when creating your tokens.

The decorated function must take **one** argument.

The argument is the identity that was used when creating a JWT.

The decorated function must return a string.

user_lookup_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to convert a JWT into a python object that can be used in a protected endpoint. This is useful for automatically loading a SQLAlchemy instance based on the contents of the JWT.

The object returned from this function can be accessed via `current_user` or `get_current_user()`

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function can return any python object, which can then be accessed in a protected endpoint. If an object cannot be loaded, for example if a user has been deleted from your database, `None` must be returned to indicate that an error occurred loading the user.

user_lookup_error_loader(*callback: Callable*) → Callable

This decorator sets the callback function used to return a custom response when loading a user via `user_lookup_loader()` fails.

The decorated function must take **two** arguments.

The first argument is a dictionary containing the header data of the JWT.

The second argument is a dictionary containing the payload data of the JWT.

The decorated function must return a Flask Response.

AFL.automation.APIServer.APIServer.LoggerFilter

class AFL.automation.APIServer.APIServer.**LoggerFilter**(**filters*)

Bases: `object`

__init__(**filters*)

Methods

__init__(*filters)	

__init__(**filters*)

AFL.automation.APIServer.APIServer.MutableQueue

class AFL.automation.APIServer.APIServer.**MutableQueue**

Bases: object

Thread-safe, mutable queue

Unlike the standard library CPython queue, this class supportes positional inserts, deletions, and reordering. The tradeoff is performance for both reads and writes to the queue.

__init__()

Methods

__init__()	
empty()	
get([loc, block, timeout])	Get next item from queue
iterationid()	
move(old_index[, new_index])	Move item in queue
put(item, loc)	Insert an item at the top of the queue
qsize()	
remove(loc)	Remove an item from the queue

__init__()

qsize()

iterationid()

empty()

put(*item*, *loc*)

Insert an item at the top of the queue

remove(*loc*)

Remove an item from the queue

get(*loc=0*, *block=True*, *timeout=None*)

Get next item from queue

move(*old_index*, *new_index=None*)

Move item in queue

AFL.automation.APIServer.APIServer.QueueDaemon

class AFL.automation.APIServer.APIServer.**QueueDaemon**(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

Bases: Thread

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(app, driver, task_queue, history[, ...])	This constructor should always be called with keyword arguments.
check_if_paused()	
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
mask_serialized_objs(package)	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

check_if_paused()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

mask_serialized_objs(*package*)

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.APIServer.APIServer.SMTPHandler

class AFL.automation.APIServer.APIServer.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=5.0*)

Bases: Handler

A handler class which sends an SMTP email for each logging event.

__init__(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=5.0*)

Initialize the handler.

Initialize the instance with the from and to addresses and subject line of the email. To specify a non-standard SMTP port, use the (host, port) tuple format for the mailhost argument. To specify authentication credentials, supply a (username, password) tuple for the credentials argument. To specify the use of a secure protocol (TLS), pass in a tuple for the secure argument. This will only be used when authentication credentials are supplied. The tuple will be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *starttls* method). A timeout in seconds can be specified for the SMTP connection (the default is one second).

Methods

__init__(mailhost, fromaddr, toaddrs, subject)	Initialize the handler.
acquire()	Acquire the I/O thread lock.
addFilter(filter)	Add the specified filter to this handler.
close()	Tidy up any resources used by the handler.
createLock()	Acquire a thread lock for serializing access to the underlying I/O.
emit(record)	Emit a record.
filter(record)	Determine if a record is loggable by consulting all the filters.
flush()	Ensure all logging output has been flushed.
format(record)	Format the specified record.
getSubject(record)	Determine the subject for the email.
get_name()	
handle(record)	Conditionally emit the specified logging record.
handleError(record)	Handle errors which occur during an emit() call.
release()	Release the I/O thread lock.
removeFilter(filter)	Remove the specified filter from this handler.
setFormatter(fmt)	Set the formatter for this handler.
setLevel(level)	Set the logging level of this handler.
set_name(name)	

Attributes

name

__init__(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=5.0*)

Initialize the handler.

Initialize the instance with the from and to addresses and subject line of the email. To specify a non-standard SMTP port, use the (host, port) tuple format for the mailhost argument. To specify authentication credentials, supply a (username, password) tuple for the credentials argument. To specify the use of a secure protocol (TLS), pass in a tuple for the secure argument. This will only be used when authentication credentials are supplied. The tuple will be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *starttls* method). A timeout in seconds can be specified for the SMTP connection (the default is one second).

getSubject(*record*)

Determine the subject for the email.

If you want to specify a subject line which is record-dependent, override this method.

emit(*record*)

Emit a record.

Format the record and send it to the specified addressees.

acquire()

Acquire the I/O thread lock.

addFilter(*filter*)

Add the specified filter to this handler.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()

Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)

Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(*record*)

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

property name

release()

Release the I/O thread lock.

removeFilter(*filter*)

Remove the specified filter from this handler.

setFormatter(*fmt*)

Set the formatter for this handler.

setLevel(*level*)

Set the logging level of this handler. level must be an int or a str.

set_name(*name*)

AFL.automation.APIServer.APIServer.ServiceInfo

class AFL.automation.APIServer.APIServer.**ServiceInfo**

Bases: RecordUpdateListener

Service information.

Constructor parameters are as follows:

- *type_*: fully qualified service type name
- *name*: fully qualified service name
- *port*: port that the service runs on
- *weight*: weight of the service
- *priority*: priority of the service
- *properties*: dictionary of properties (or a bytes object holding the contents of the *text* field). converted to str and then encoded to bytes using UTF-8. Keys with *None* values are converted to value-less attributes.
- *server*: fully qualified name for service host (defaults to name)
- *host_ttl*: ttl used for A/SRV records
- *other_ttl*: ttl used for PTR/TXT records
- *addresses* and *parsed_addresses*: List of IP addresses (either as bytes, network byte order, or in parsed form as text; at most one of those parameters can be provided)
- interface_index: scope_id or zone_id for IPv6 link-local addresses i.e. an identifier of the interface where the peer is connected to

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
addresses_by_version(version)	List addresses matching IP version.
async_clear_cache()	Clear the cache for this service info.
async_request(zc, timeout[, question_type, ...])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
async_update_records(zc, now, records)	Updates service information from a DNS record.
async_update_records_complete()	Called when a record update has completed for all handlers.
async_wait(timeout[, loop])	Calling task waits for a given number of milliseconds or until notified.
dns_addresses([override_ttl, version])	Return matching DNSAddress from ServiceInfo.
dns_nsec(missing_types[, override_ttl])	Return DNSNsec from ServiceInfo.
dns_pointer([override_ttl])	Return DNSPointer from ServiceInfo.
dns_service([override_ttl])	Return DNSService from ServiceInfo.
dns_text([override_ttl])	Return DNSText from ServiceInfo.
get_address_and_nsec_records([override_ttl])	Build a set of address records and NSEC records for non-present record types.
get_name()	Name accessor
ip_addresses_by_version(version)	List ip_address objects matching IP version.
load_from_cache(zc[, now])	Populate the service info from the cache.
parsed_addresses([version])	List addresses in their parsed string form.
parsed_scoped_addresses([version])	Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available
request(zc, timeout[, question_type, addr, port])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
set_server_if_missing()	Set the server if it is missing.
update_record(zc, now, record)	Update a single record.

Attributes

host_ttl	
interface_index	
key	
other_ttl	
port	
priority	
server	
server_key	
text	
type	
weight	
addresses	IPv4 addresses of this service.
decoded_properties	Return properties as strings.
name	The name of the service.
properties	Return properties as bytes.

host_ttl

interface_index

key

other_ttl

port

priority

server

server_key

text

type

weight

__init__(**args*, ***kwargs*)

addresses

IPv4 addresses of this service.

Only IPv4 addresses are returned for backward compatibility. Use *addresses_by_version()* or *parsed_addresses()* to include IPv6 addresses as well.

addresses_by_version(*version:* IPVersion)

List addresses matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

async_clear_cache()

Clear the cache for this service info.

async async_request(*zc:* Zeroconf, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

This method will be run in the event loop.

Passing addr and port is optional, and will default to the mDNS multicast address and port. This is useful for directing requests to a specific host that may be able to respond across subnets.

Parameters

- **zc** – Zeroconf instance
- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

async_update_records(*zc:* Zeroconf, *now: float_*, *records: list[RecordUpdate]*)

Updates service information from a DNS record.

This method will be run in the event loop.

async_update_records_complete()

Called when a record update has completed for all handlers.

At this point the cache will have the new records.

This method will be run in the event loop.

async async_wait(*timeout: float*, *loop: AbstractEventLoop | None = None*) → None

Calling task waits for a given number of milliseconds or until notified.

decoded_properties

Return properties as strings.

dns_addresses(*override_ttl: int_ | None = None*, *version:* IPVersion *= IPVersion.All*) → list[DNSAddress]

Return matching DNSAddress from ServiceInfo.

dns_nsec(*missing_types: list[int]*, *override_ttl: int_ | None = None*) → DNSNsec

Return DNSNsec from ServiceInfo.

dns_pointer(*override_ttl: int_ | None = None*) → DNSPointer

Return DNSPointer from ServiceInfo.

dns_service(*override_ttl: int_ | None = None*) → DNSService

Return DNSService from ServiceInfo.

dns_text(*override_ttl: int_ | None = None*) → DNSText

Return DNSText from ServiceInfo.

get_address_and_nsec_records(*override_ttl: int_ | None = None*) → set[DNSRecord]

Build a set of address records and NSEC records for non-present record types.

get_name() → str

Name accessor

ip_addresses_by_version(*version: IPVersion*)

List ip_address objects matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

load_from_cache(*zc: Zeroconf*, *now: float_ | None = None*) → bool

Populate the service info from the cache.

This method is designed to be threadsafe.

name

The name of the service.

parsed_addresses(*version: IPVersion = IPVersion.All*) → list[str]

List addresses in their parsed string form.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

parsed_scoped_addresses(*version: IPVersion = IPVersion.All*) → list[str]

Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

properties

Return properties as bytes.

request(*zc: Zeroconf*, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_request* cannot be completed.

Parameters

- **zc** – Zeroconf instance

- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

set_server_if_missing() → None

Set the server if it is missing.

This function is for backwards compatibility.

update_record(*zc:* Zeroconf, *now: float*, *record: DNSRecord*) → None

Update a single record.

This method is deprecated and will be removed in a future version. update_records should be implemented instead.

AFL.automation.APIServer.APIServer.Zeroconf

class AFL.automation.APIServer.APIServer.**Zeroconf**(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version:* IPVersion | *None = None*, *apple_p2p: bool = False*)

Bases: QuietLogger

Implementation of Zeroconf Multicast DNS Service Discovery

Supports registration, unregistration, queries and browsing.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version:* IPVersion | *None = None*, *apple_p2p: bool = False*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.

Parameters

- **interfaces** – InterfaceChoice or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (::1) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.
- **apple_p2p** – use AWDL interface (only macOS)

Methods

__init__([interfaces, unicast, ip_version, ...])	Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.
add_listener(listener, question)	Adds a listener for a given question.

continues on next page

Table 4 – continued from previous page

add_service_listener(type_, listener)	Adds a listener for a particular service type.
async_add_listener(listener, question)	Adds a listener for a given question.
async_check_service(info, allow_name_change)	Checks the network for a unique service name, modifying the ServiceInfo passed in if it is not unique.
async_get_service_info(type_, name[, ...])	Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.
async_notify_all()	Schedule an async_notify_all.
async_register_service(info[, ttl, ...])	Registers service information to the network with a default TTL.
async_remove_listener(listener)	Removes a listener.
async_send(out[, addr, port, v6_flow_scope, ...])	Sends an outgoing packet.
async_unregister_all_services()	Unregister all registered services.
async_unregister_service(info)	Unregister a service.
async_update_service(info)	Registers service information to the network with a default TTL.
async_wait(timeout)	Calling task waits for a given number of milliseconds or until notified.
async_wait_for_start([timeout])	Wait for start up for actions that require a running Zeroconf instance.
close()	Ends the background threads, and prevent this instance from servicing further queries.
generate_service_broadcast(info, ttl[, ...])	Generate a broadcast to announce a service.
generate_service_query(info)	Generate a query to lookup a service.
generate_unregister_all_services()	Generate a DNSOutgoing goodbye for all services and remove them from the registry.
get_service_info(type_, name[, timeout, ...])	Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.
log_exception_debug(*logger_data)	
log_exception_once(exc, *args)	
log_exception_warning(*logger_data)	
log_warning_once(*args)	
notify_all()	Notifies all waiting threads and notify listeners.
register_service(info[, ttl, ...])	Registers service information to the network with a default TTL.
remove_all_service_listeners()	Removes a listener from the set that is currently listening.
remove_listener(listener)	Removes a listener.
remove_service_listener(listener)	Removes a listener from the set that is currently listening.
send(out[, addr, port, v6_flow_scope, transport])	Sends an outgoing packet threadsafe.
start()	Start Zeroconf.
unregister_all_services()	Unregister all registered services.
unregister_service(info)	Unregister a service.
update_service(info)	Registers service information to the network with a default TTL.

Attributes

listeners	
started	Check if the instance has started.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version: IPVersion | None = None*, *apple_p2p: bool = False*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.

Parameters

- **interfaces** – `InterfaceChoice` or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (`::1`) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.
- **apple_p2p** – use AWDL interface (only macOS)

property started: bool

Check if the instance has started.

start() → None

Start Zeroconf.

async async_wait_for_start(*timeout: float = 9*) → None

Wait for start up for actions that require a running Zeroconf instance.

Throws NotRunningException if the instance is not running or could not be started.

property listeners: set[RecordUpdateListener]

async async_wait(*timeout: float*) → None

Calling task waits for a given number of milliseconds or until notified.

notify_all() → None

Notifies all waiting threads and notify listeners.

async_notify_all() → None

Schedule an async_notify_all.

get_service_info(*type_: str*, *name: str*, *timeout: int = 3000*, *question_type: DNSQuestionType | None = None*) → *ServiceInfo* | None

Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.

Parameters

- **type** – fully qualified service type name

- **name** – the name of the service
- **timeout** – milliseconds to wait for a response
- **question_type** – The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

add_service_listener(*type_: str*, *listener: ServiceListener*) → None

Adds a listener for a particular service type. This object will then have its add_service and remove_service methods called when services of that type become available and unavailable.

remove_service_listener(*listener: ServiceListener*) → None

Removes a listener from the set that is currently listening.

remove_all_service_listeners() → None

Removes a listener from the set that is currently listening.

register_service(*info:* ServiceInfo, *ttl: int | None = None*, *allow_name_change: bool = False*, *cooperating_responders: bool = False*, *strict: bool = True*) → None

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service. The name of the service may be changed if needed to make it unique on the network. Additionally multiple cooperating responders can register the same service on the network for resilience (if you want this behavior set *cooperating_responders* to *True*).

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *register_service* cannot be completed.

async async_register_service(*info:* ServiceInfo, *ttl: int | None = None*, *allow_name_change: bool = False*, *cooperating_responders: bool = False*, *strict: bool = True*) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service. The name of the service may be changed if needed to make it unique on the network. Additionally multiple cooperating responders can register the same service on the network for resilience (if you want this behavior set *cooperating_responders* to *True*).

update_service(*info:* ServiceInfo) → None

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_update_service* cannot be completed.

async async_update_service(*info:* ServiceInfo) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service.

async async_get_service_info(*type_: str*, *name: str*, *timeout: int = 3000*, *question_type: DNSQuestionType | None = None*) → *AsyncServiceInfo* | None

Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.

Parameters

- **type** – fully qualified service type name
- **name** – the name of the service
- **timeout** – milliseconds to wait for a response
- **question_type** – The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

generate_service_broadcast(*info:* ServiceInfo, *ttl: int | None*, *broadcast_addresses: bool = True*) → DNSOutgoing

Generate a broadcast to announce a service.

generate_service_query(*info:* ServiceInfo) → DNSOutgoing

Generate a query to lookup a service.

unregister_service(*info:* ServiceInfo) → None

Unregister a service.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_unregister_service* cannot be completed.

async async_unregister_service(*info:* ServiceInfo) → Awaitable

Unregister a service.

generate_unregister_all_services() → DNSOutgoing | None

Generate a DNSOutgoing goodbye for all services and remove them from the registry.

async async_unregister_all_services() → None

Unregister all registered services.

Unlike async_register_service and async_unregister_service, this method does not return a future and is always expected to be awaited since its only called at shutdown.

unregister_all_services() → None

Unregister all registered services.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_unregister_all_services* cannot be completed.

async async_check_service(*info:* ServiceInfo, *allow_name_change: bool*, *cooperating_responders: bool = False*, *strict: bool = True*) → None

Checks the network for a unique service name, modifying the ServiceInfo passed in if it is not unique.

add_listener(*listener: RecordUpdateListener*, *question: DNSQuestion | list[DNSQuestion] | None*) → None

Adds a listener for a given question. The listener will have its update_record method called when information is available to answer the question(s).

This function is threadsafe

remove_listener(*listener: RecordUpdateListener*) → None

Removes a listener.

This function is threadsafe

async_add_listener(*listener: RecordUpdateListener*, *question: DNSQuestion | list[DNSQuestion] | None*) → None

Adds a listener for a given question. The listener will have its update_record method called when information is available to answer the question(s).

This function is not threadsafe and must be called in the eventloop.

async_remove_listener(*listener: RecordUpdateListener*) → None

Removes a listener.

This function is not threadsafe and must be called in the eventloop.

send(*out: DNSOutgoing*, *addr: str | None = None*, *port: int = 5353*, *v6_flow_scope: tuple[()] | tuple[int, int] = ()*, *transport: _WrappedTransport | None = None*) → None

Sends an outgoing packet threadsafe.

async_send(*out: DNSOutgoing*, *addr: str | None = None*, *port: int = 5353*, *v6_flow_scope: tuple[()] | tuple[int, int] = ()*, *transport: _WrappedTransport | None = None*) → None

Sends an outgoing packet.

close() → None

Ends the background threads, and prevent this instance from servicing further queries.

This method is idempotent and irreversible.

classmethod log_exception_debug(**logger_data: Any*) → None

classmethod log_exception_once(*exc: Exception*, **args: Any*) → None

classmethod log_exception_warning(**logger_data: Any*) → None

classmethod log_warning_once(**args: Any*) → None

class AFL.automation.APIServer.APIServer.**APIServer**(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

__init__(*name*, *data=None*, *experiment='Development'*, *contact='tbm@nist.gov'*, *index_template='index.html'*, *new_index_template='index-new.html'*, *plot_template='simple-bokeh.html'*)

create_queue(*driver*, *add_unqueued=True*)

reset_queue_daemon(*driver=None*)

advertise_zeroconf(***kwargs*)

run(***kwargs*)

run_threaded(*start_thread=True*, ***kwargs*)

add_standard_routes()

get_info()

Live, status page of the robot

get_quickbar()

Return the functions, params, and defaults to be shown in this server's quickbar

is_server_live()

get_unqueued_commands()

get_queued_commands()

add_unqueued_routes()

query_driver()

init_logging(*toaddrs=None*)

index()

Live, status page of the robot

index_new()

Live, status page of the robot

webapp()

Live, status page of the robot

render_unqueued(*func*, *kwargs_add*, ***kwargs*)

Convert an unqueued return item into web-suitable output

send_1d_plot(*result*, *multi=False*, ***kwargs*)

send_array_as_jpg(*array*, *log_image=False*, *max_val=None*, *fillna=0.0*, ***kwargs*)

queue_state()

driver_status()

get_queue()

get_queue_iteration()

deposit_obj()

Store an object named obj in the driver's dropbox If a uuid is provided, the object will be stored with that uuid Otherwise, a new uuid will be generated. In either case, the uuid will be returned to the client.

retrieve_obj()

Retrieve an object from the driver's dropbox uuid specifies the object to retrieve delete specifies whether to delete the object after retrieval

set_driver_object()

get_driver_object()

enqueue()

reorder_queue()

remove_items()

remove_item()

move_item()

clear_queue()

clear_history()

debug()

pause()

halt()

init()

login()

get_server_time()

login_test()

AFL.automation.APIServer.Client

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
ServerDiscovery()	ServerDiscovery class

AFL.automation.APIServer.Client.Client

class AFL.automation.APIServer.Client.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	

continues on next page

Table 5 – continued from previous page

get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.APIServer.Client.ServerDiscovery

class AFL.automation.APIServer.Client.**ServerDiscovery**

Bases: object

ServerDiscovery class

This class is used to discover AFL servers on the network using zeroconf requests that match a particular specification.

__init__()

Methods

__init__()	
aio_find_server_by_name(service_name)	
discover_server_by_name(service_name)	Does a zeroconf request to find a named AFL-automation server on the network.
find_server_by_name(service_name)	Disambiguator for either matching or discovering a specific server by name
find_server_by_partial_name(service_name)	Looks through the registry of discovered services for a partial name match.
find_server_by_property_match(property_name ...)	Looks through the registry of discovered services for a partial match in a property string.
get_service_info(zeroconf, service_type, name)	
manage_service_info_to_list(zeroconf, ...)	
match_server_by_name(service_name)	Looks through the registry of discovered services for an exact name match.
on_service_state_change(zeroconf, ...)	
sa_aio_discover_server_by_name(service_name	Does a zeroconf request to find a named AFL-automation server on the network.
sa_discover_server_by_name(service_name)	Does a zeroconf request to find a named AFL-automation server on the network.

__init__()

on_service_state_change(*zeroconf: Zeroconf*, *service_type: str*, *name: str*, *state_change: ServiceStateChange*) → None

async manage_service_info_to_list(*zeroconf*, *service_type*, *name*, *append_or_remove*)

async get_service_info(*zeroconf: Zeroconf*, *service_type: str*, *name: str*) → None

async aio_find_server_by_name(*service_name*)

discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

async classmethod sa_aio_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

classmethod sa_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_name(*service_name*)

Disambiguator for either matching or discovering a specific server by name

match_server_by_name(*service_name*)

Looks through the registry of discovered services for an exact name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_partial_name(*service_name*)

Looks through the registry of discovered services for a partial name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_property_match(*property_name*, *property_value*)

Looks through the registry of discovered services for a partial match in a property string. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

class AFL.automation.APIServer.Client.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.APIServer.Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
makeRegistrar()	
sqrt(x, /)	Return the square root of x.

AFL.automation.APIServer.Driver.ceil

AFL.automation.APIServer.Driver.**ceil**(*x*, */*)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

AFL.automation.APIServer.Driver.listify

AFL.automation.APIServer.Driver.**listify**(*obj*)

AFL.automation.APIServer.Driver.makeRegistrar

AFL.automation.APIServer.Driver.**makeRegistrar**()

AFL.automation.APIServer.Driver.sqrt

AFL.automation.APIServer.Driver.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
PersistentConfig(path[, defaults, ...])	A dictionary-like class that serializes changes to disk

AFL.automation.APIServer.Driver.Driver

class AFL.automation.APIServer.Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.APIServer.Driver.PersistentConfig

class AFL.automation.APIServer.Driver.**PersistentConfig**(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Bases: *MutableMapping*

A dictionary-like class that serializes changes to disk

This class provides dictionary-like setters and getters (e.g., [] and .update()) but, by default, all modifications are written into a file in json format with the root keys as timestamps. Modifications can be blocked by locking the config, and writing to disk can be disabled by setting the appropriate member attributes (see constructor). On instantiation, if provided with a previously saved configuration file, PersistentConfig will load the file's contents into memory and use the more recent configuration.

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

Methods

__init__(path[, defaults, overrides, lock, ...])	Constructor
clear()	
get(k[,d])	
get_historical_values(key[, convert_to_datetime])	Convenience method for gathering historical values of a parameter
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
revert([nth, datetime_key])	Revert config to a historical config
setdefault(k[,d])	
toJSON()	Serialize the config to json
update(update_dict)	Update several values in config at once
values()	

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

__getitem__(*key*)

Dictionary-like getter via config["param"]

__setitem__(*key*, *value*)

Dictionary-like setter via config["param"]=value

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

toJSON()

Serialize the config to json

update(*update_dict*)

Update several values in config at once

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

revert(*nth=None*, *datetime_key=None*)

Revert config to a historical config

Parameters

- **nth** (int, **optional***) – Integer index of historical value to revert to. Can be negative to count from end of history. Note that -1 will correspond to the current config.
- **datetime_key** (str, **optional**) – datetime formatted string as defined by datetime_key_format

get_historical_values(*key*, *convert_to_datetime=False*)

Convenience method for gathering historical values of a parameter

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

values() → an object providing a view on D's values

AFL.automation.APIServer.Driver.**makeRegistrar**()

class AFL.automation.APIServer.Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()
Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)
Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)
Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.APIServer.DummyDriver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

AFL.automation.APIServer.DummyDriver.ceil

AFL.automation.APIServer.DummyDriver.**ceil**(*x*, */*)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

AFL.automation.APIServer.DummyDriver.listify

AFL.automation.APIServer.DummyDriver.**listify**(*obj*)

AFL.automation.APIServer.DummyDriver.sqrt

AFL.automation.APIServer.DummyDriver.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
DummyDriver([name, overrides])	

AFL.automation.APIServer.DummyDriver.Driver

class AFL.automation.APIServer.DummyDriver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.APIServer.DummyDriver.DummyDriver

class AFL.automation.APIServer.DummyDriver.**DummyDriver**(*name=None*, *overrides=None*)

Bases: *Driver*

__init__(*name=None*, *overrides=None*)

Methods

__init__([name, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
dummy_reset_tank_levels([rinse1, rinse2, waste])	
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
how_many(**kwargs)	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
quickbar_test([text_field, int_field, ...])	
quickbar_test2()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
test_command1([kwarg1, kwarg2])	A test command with positional and keyword parameters
test_command2([kwarg1, kwarg2, kwarg3])	A test command with positional and keyword parameters
test_command_sets_data([kwarg1, kwarg2, kwarg3])	A test command with positional and keyword parameters
test_image(**kwargs)	
test_plot(**kwargs)	
unqueued()	

Attributes

defaults

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

test_command1(*kwarg1=None*, *kwarg2=True*)

A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.APIServer.DummyDriver.**DummyDriver**(*name=None*, *overrides=None*)

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

test_command1(*kwarg1=None*, *kwarg2=True*)

A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

AFL.automation.APIServer.DummyOT2Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

AFL.automation.APIServer.DummyOT2Driver.ceil

AFL.automation.APIServer.DummyOT2Driver.**ceil**(*x*, */*)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

AFL.automation.APIServer.DummyOT2Driver.listify

AFL.automation.APIServer.DummyOT2Driver.**listify**(*obj*)

AFL.automation.APIServer.DummyOT2Driver.sqrt

AFL.automation.APIServer.DummyOT2Driver.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
DummyDriver([name, overrides])	

AFL.automation.APIServer.DummyOT2Driver.Driver

class AFL.automation.APIServer.DummyOT2Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.APIServer.DummyOT2Driver.DummyDriver

class AFL.automation.APIServer.DummyOT2Driver.**DummyDriver**(*name=None*, *overrides=None*)

Bases: *Driver*

__init__(*name=None*, *overrides=None*)

Methods

__init__([name, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
dummy_reset_tank_levels([rinse1, rinse2, waste])	
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_prep_target(**kwargs)	
get_sample()	
how_many(**kwargs)	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
quickbar_test([text_field, int_field, ...])	
quickbar_test2()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
test_command1([kwarg1, kwarg2])	A test command with positional and keyword parameters
test_command2([kwarg1, kwarg2, kwarg3])	A test command with positional and keyword parameters
test_command_sets_data([kwarg1, kwarg2, kwarg3])	A test command with positional and keyword parameters

continues on next page

Table 6 – continued from previous page

test_image(**kwargs)
test_plot(**kwargs)
unqueued()

Attributes

defaults

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

execute(***kwargs*)

get_prep_target(***kwargs*)

test_command1(*kwarg1=None*, *kwarg2=True*)

A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)

A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.APIServer.DummyOT2Driver.**DummyDriver**(*name=None*, *overrides=None*)

defaults = {'density of water': 1.0, 'speed of light': 300000000.0}

__init__(*name=None*, *overrides=None*)

status()

execute(***kwargs*)

get_prep_target(***kwargs*)

test_command1(*kwarg1=None*, *kwarg2=True*)
A test command with positional and keyword parameters

test_command2(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)
A test command with positional and keyword parameters

test_command_sets_data(*kwarg1=False*, *kwarg2=False*, *kwarg3=True*)
A test command with positional and keyword parameters

how_many(***kwargs*)

test_plot(***kwargs*)

test_image(***kwargs*)

quickbar_test(*text_field='Three'*, *int_field=3*, *float_field=3.14*, *bool_field=True*)

quickbar_test2()

dummy_reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

AFL.automation.APIServer.LoggerFilter

Classes

LoggerFilter(*filters)

AFL.automation.APIServer.LoggerFilter.LoggerFilter

class AFL.automation.APIServer.LoggerFilter.**LoggerFilter**(**filters*)
Bases: object

__init__(**filters*)

Methods

__init__(*filters)

__init__(**filters*)

class AFL.automation.APIServer.LoggerFilter.**LoggerFilter**(**filters*)

__init__(**filters*)

AFL.automation.APIServer.QueueDaemon

Functions

is_serialized(obj)

AFL.automation.APIServer.QueueDaemon.is_serialized

AFL.automation.APIServer.QueueDaemon.**is_serialized**(*obj*)

Classes

DataTrashcan()	A DataPacket implementation *for testing only* that takes all its data and simply throws it away.
QueueDaemon(app, driver, task_queue, history)	

AFL.automation.APIServer.QueueDaemon.DataTrashcan

class AFL.automation.APIServer.QueueDaemon.**DataTrashcan**

Bases: DataPacket

A DataPacket implementation *for testing only* that takes all its data and simply throws it away.

__init__()

Methods

__init__()	
add_array(*args, **kwargs)	Abstract method adding arrays that need special handling
clear()	
finalize()	
get(k[,d])	
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
reset()	Clears all transient data.
reset_sample()	
setdefault(k[,d])	
setupDefaults()	
transmit()	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

Attributes

PROTECTED_SAMPLE_KEYS
PROTECTED_SYSTEM_KEYS

transmit()

add_array(**args*, ***kwargs*)

Abstract method adding arrays that need special handling

PROTECTED_SAMPLE_KEYS = ['sample_name', 'sample_uuid', 'sample_composition', 'AL_components', 'AL_campaign_name', 'AL_uuid']

PROTECTED_SYSTEM_KEYS = ['driver_name', 'driver_config', 'platform_serial']

__init__()

clear() → None. Remove all items from D.

finalize()

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

reset()
Clears all transient data.

reset_sample()

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

setupDefaults()

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

AFL.automation.APIServer.QueueDaemon.QueueDaemon

class AFL.automation.APIServer.QueueDaemon.**QueueDaemon**(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

Bases: Thread

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(app, driver, task_queue, history[, ...])	This constructor should always be called with keyword arguments.
check_if_paused()	
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
mask_serialized_objs(package)	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

check_if_paused()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

mask_serialized_objs(*package*)

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.APIServer.QueueDaemon.**QueueDaemon**(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

__init__(*app*, *driver*, *task_queue*, *history*, *debug=False*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

check_if_paused()

mask_serialized_objs(*package*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

6.2.2 Instrument

The Instrument module contains drivers and interfaces for various scientific instruments.

AFL.automation.instrument

AFL.automation.instrument

Modules

DummySAS
FileCamera
I22SAXS
NetworkCamera
SeabreezeUVVis
SpecScreen_Driver

AFL.automation.instrument.DummySAS

Classes

Driver(name[, defaults, overrides])
DummySAS([overrides])

AFL.automation.instrument.DummySAS.Driver

class AFL.automation.instrument.DummySAS.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.instrument.DummySAS.DummySAS

class AFL.automation.instrument.DummySAS.**DummySAS**(*overrides=None*)

Bases: *Driver*

__init__(*overrides=None*)

connect to spec

Methods

__init__([overrides])	connect to spec
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
expose([name, exposure, nexp, block, ...])	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

Attributes

defaults

defaults = {}

__init__(*overrides=None*)

connect to spec

expose(*name=None*, *exposure=None*, *nexp=1*, *block=True*, *reduce_data=True*, *measure_transmission=True*, *save_nexus=True*)

status()

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.instrument.DummySAS.**DummySAS**(*overrides=None*)

defaults = {}

__init__(*overrides=None*)
connect to spec

expose(*name=None*, *exposure=None*, *nexp=1*, *block=True*, *reduce_data=True*, *measure_transmission=True*, *save_nexus=True*)

status()

AFL.automation.instrument.FileCamera

Classes

FileCamera()

AFL.automation.instrument.FileCamera.FileCamera

class AFL.automation.instrument.FileCamera.**FileCamera**
Bases: object

__init__()

Methods

__init__()
collect(fname)

__init__()

collect(*fname*)

class AFL.automation.instrument.FileCamera.**FileCamera**

__init__()

collect(*fname*)

AFL.automation.instrument.I22SAXS

Classes

Driver(name[, defaults, overrides])
I22SAXS([overrides])

AFL.automation.instrument.I22SAXS.Driver

class AFL.automation.instrument.I22SAXS.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.instrument.I22SAXS.I22SAXS

class AFL.automation.instrument.I22SAXS.**I22SAXS**(*overrides=None*, ***kwargs*)

Bases: *Driver*

__init__(*overrides=None*, ***kwargs*)

Methods

__init__([overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
expose(name[, empty, nframes, acq_time, ...])	Perform a sequence of exposures at positions defined in self.config['pos_list'].
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
read_integrated(scanid)	Scans the appropriate directory for the filename of the data collected with filename
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

Attributes

defaults

defaults = {'acq_time': 1, 'acq_timeout': 120, 'address': '', 'data_read_cooldown': 5, 'empty_scan_id': '', 'file_read_timeout': 30, 'nframes': 1, 'port': 2222, 'pos_list': [], 'processed_base_path': '/mnt/i22_processed/i22-', 'reduced_data_suffix': '_saxs_Transmission_Averaged_Subtracted_IvsQ_processed.nxs', 'ssh_key_path': '', 'username': ''}

__init__(*overrides=None*, ***kwargs*)

expose(*name*, *empty=False*, *nframes=None*, *acq_time=None*, *set_empty=False*)

Perform a sequence of exposures at positions defined in self.config['pos_list'].

Return the integrated data of the measurement with the lowest measured sample transmission.

read_integrated(*scanid*)

Scans the appropriate directory for the filename of the data collected with filename

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

status()

unqueued()

class AFL.automation.instrument.I22SAXS.**I22SAXS**(*overrides=None*, ***kwargs*)

defaults = {'acq_time': 1, 'acq_timeout': 120, 'address': '', 'data_read_cooldown': 5, 'empty_scan_id': '', 'file_read_timeout': 30, 'nframes': 1, 'port': 2222, 'pos_list': [], 'processed_base_path': '/mnt/i22_processed/i22-', 'reduced_data_suffix': '_saxs_Transmission_Averaged_Subtracted_IvsQ_processed.nxs', 'ssh_key_path': '', 'username': ''}

__init__(*overrides=None*, ***kwargs*)

expose(*name*, *empty=False*, *nframes=None*, *acq_time=None*, *set_empty=False*)

Perform a sequence of exposures at positions defined in self.config[‘pos_list’].

Return the integrated data of the measurement with the lowest measured sample transmission.

read_integrated(*scanid*)

Scans the appropriate directory for the filename of the data collected with filename

AFL.automation.instrument.NetworkCamera

Classes

NetworkCamera(url)

AFL.automation.instrument.NetworkCamera.NetworkCamera

class AFL.automation.instrument.NetworkCamera.**NetworkCamera**(*url*)

Bases: object

__init__(*url*)

Methods

__init__(url)	
camera_reset()	
collect()	

__init__(*url*)

camera_reset()

collect()

class AFL.automation.instrument.NetworkCamera.**NetworkCamera**(*url*)

__init__(*url*)

camera_reset()

collect()

AFL.automation.instrument.SeabreezeUVVis

Classes

Driver(name[, defaults, overrides])	
Eq(key, value)	Query equality of a given key's value to the specified value.
Path(*args, **kwargs)	PurePath subclass that can make system calls.
SeabreezeUVVis([backend, device_serial, ...])	

AFL.automation.instrument.SeabreezeUVVis.Driver

class AFL.automation.instrument.SeabreezeUVVis.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.instrument.SeabreezeUVVis.Eq

class AFL.automation.instrument.SeabreezeUVVis.**Eq**(*key: str*, *value: Any*)

Bases: NoBool

Query equality of a given key's value to the specified value.

See *Key* in this module for a more intuitive interface for equality.

Parameters

- **key** (*str*) – e.g. “color”, “sample.name”
- **value** (*JSONSerializable*) – May be a string, number, list, or dict.

Examples

Search for color == “red”

```
>>> c.search(Eq("color", "red"))
```

__init__(*key: str*, *value: Any*) → None

Methods

__init__(key, value)
decode(*, key, value)
encode()

Attributes

key
value

key: str

value: Any

encode()

classmethod decode(*, *key*, *value*)

__init__(*key: str*, *value: Any*) → None

AFL.automation.instrument.SeabreezeUVVis.Path

class AFL.automation.instrument.SeabreezeUVVis.**Path**(**args*, ***kwargs*)

Bases: PurePath

PurePath subclass that can make system calls.

Path represents a filesystem path but unlike PurePath, also offers methods to do system calls on path objects. Depending on your system, instantiating a Path will return either a PosixPath or a WindowsPath object. You can also instantiate a PosixPath or WindowsPath directly, but cannot instantiate a WindowsPath on a POSIX system or vice versa.

__init__()

Methods

__init__()	
absolute()	Return an absolute version of this path by prepending the current working directory.
as_posix()	Return the string representation of the path with forward (/) slashes.
as_uri()	Return the path as a 'file' URI.
chmod(mode, *[, follow_symlinks])	Change the permissions of the path, like os.chmod().
cwd()	Return a new path pointing to the current working directory (as returned by os.getcwd()).
exists()	Whether this path exists.
expanduser()	Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)
glob(pattern)	Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.
group()	Return the group name of the file gid.
hardlink_to(target)	Make this path a hard link pointing to the same file as *target*.
home()	Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.
is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_mount()	Check if this path is a POSIX mount point
is_relative_to(*other)	Return True if the path is relative to another path or False.
is_reserved()	Return True if the path contains one of the special names reserved by the system, if any.
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.
iterdir()	Iterate over the files in this directory.
joinpath(*args)	Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).
lchmod(mode)	Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
link_to(target)	Make the target path a hard link pointing to this path.
lstat()	Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.
match(path_pattern)	Return True if this path matches the given pattern.
mkdir([mode, parents, exist_ok])	Create a new directory at this given path.

continues on next page

Table 7 – continued from previous page

open([mode, buffering, encoding, errors, ...])	Open the file pointed by this path and return a file object, as the built-in open() function does.
owner()	Return the login name of the file owner.
read_bytes()	Open the file in bytes mode, read it, and close the file.
read_text([encoding, errors])	Open the file in text mode, read it, and close the file.
readlink()	Return the path to which the symbolic link points.
relative_to(*other)	Return the relative path to another path identified by the passed arguments.
rename(target)	Rename this path to the target path.
replace(target)	Rename this path to the target path, overwriting if that path exists.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it.
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
rmdir()	Remove this directory.
samefile(other_path)	Return whether other_path is the same or not as this file (as returned by os.path.samefile()).
stat(*[, follow_symlinks])	Return the result of the stat() system call on this path, like os.stat() does.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the target path.
touch([mode, exist_ok])	Create this file with the given access mode, if it doesn't exist.
unlink([missing_ok])	Remove this file or link.
with_name(name)	Return a new path with the file name changed.
with_stem(stem)	Return a new path with the stem changed.
with_suffix(suffix)	Return a new path with the file suffix changed.
write_bytes(data)	Open the file in bytes mode, write to it, and close the file.
write_text(data[, encoding, errors, newline])	Open the file in text mode, write to it, and close the file.

Attributes

anchor	The concatenation of the drive and root, or ''.
drive	The drive prefix (letter or UNC path), if any.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path's logical parents.
parts	An object providing sequence-like access to the components in the filesystem path.
root	The root of the path, if any.
stem	The final path component, minus its last suffix.
suffix	The final component's last suffix, if any.
suffixes	A list of the final component's suffixes, if any.

classmethod cwd()

Return a new path pointing to the current working directory (as returned by os.getcwd()).

classmethod home()

Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).

samefile(*other_path*)

Return whether other_path is the same or not as this file (as returned by os.path.samefile()).

iterdir()

Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

glob(*pattern*)

Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

rglob(*pattern*)

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

absolute()

Return an absolute version of this path by prepending the current working directory. No normalization or symlink resolution is performed.

Use resolve() to get the canonical path to a file.

resolve(*strict=False*)

Make the path absolute, resolving all symlinks on the way and also normalizing it.

stat(**, follow_symlinks=True*)

Return the result of the stat() system call on this path, like os.stat() does.

owner()

Return the login name of the file owner.

group()

Return the group name of the file gid.

open(*mode='r', buffering=-1, encoding=None, errors=None, newline=None*)

Open the file pointed by this path and return a file object, as the built-in open() function does.

read_bytes()

Open the file in bytes mode, read it, and close the file.

read_text(*encoding=None, errors=None*)

Open the file in text mode, read it, and close the file.

write_bytes(*data*)

Open the file in bytes mode, write to it, and close the file.

write_text(*data, encoding=None, errors=None, newline=None*)

Open the file in text mode, write to it, and close the file.

readlink()

Return the path to which the symbolic link points.

touch(*mode=438, exist_ok=True*)

Create this file with the given access mode, if it doesn't exist.

mkdir(*mode=511, parents=False, exist_ok=False*)

Create a new directory at this given path.

chmod(*mode*, *, *follow_symlinks=True*)

Change the permissions of the path, like os.chmod().

lchmod(*mode*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

unlink(*missing_ok=False*)

Remove this file or link. If the path is a directory, use rmdir() instead.

rmdir()

Remove this directory. The directory must be empty.

lstat()

Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.

rename(*target*)

Rename this path to the target path.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

replace(*target*)

Rename this path to the target path, overwriting if that path exists.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

symlink_to(*target*, *target_is_directory=False*)

Make this path a symlink pointing to the target path. Note the order of arguments (link, target) is the reverse of os.symlink.

hardlink_to(*target*)

Make this path a hard link pointing to the same file as *target*.

Note the order of arguments (self, target) is the reverse of os.link's.

link_to(*target*)

Make the target path a hard link pointing to this path.

Note this function does not make this path a hard link to *target*, despite the implication of the function and argument names. The order of arguments (target, link) is the reverse of Path.symlink_to, but matches that of os.link.

Deprecated since Python 3.10 and scheduled for removal in Python 3.12. Use *hardlink_to()* instead.

exists()

Whether this path exists.

is_dir()

Whether this path is a directory.

is_file()

Whether this path is a regular file (also True for symlinks pointing to regular files).

is_mount()

Check if this path is a POSIX mount point

is_symlink()

Whether this path is a symbolic link.

is_block_device()

Whether this path is a block device.

is_char_device()

Whether this path is a character device.

is_fifo()

Whether this path is a FIFO.

is_socket()

Whether this path is a socket.

expanduser()

Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)

__bytes__()

Return the bytes representation of the path. This is only recommended to use under Unix.

__str__()

Return the string representation of the path, suitable for passing to system calls.

property anchor

The concatenation of the drive and root, or ‘’.

as_posix()

Return the string representation of the path with forward (/) slashes.

as_uri()

Return the path as a ‘file’ URI.

property drive

The drive prefix (letter or UNC path), if any.

is_absolute()

True if the path is absolute (has both a root and, if applicable, a drive).

is_relative_to(**other*)

Return True if the path is relative to another path or False.

is_reserved()

Return True if the path contains one of the special names reserved by the system, if any.

joinpath(**args*)

Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).

match(*path_pattern*)

Return True if this path matches the given pattern.

property name

The final path component, if any.

property parent

The logical parent of the path.

property parents

A sequence of this path's logical parents.

property parts

An object providing sequence-like access to the components in the filesystem path.

relative_to(**other*)

Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

property root

The root of the path, if any.

property stem

The final path component, minus its last suffix.

property suffix

The final component's last suffix, if any.

This includes the leading period. For example: '.txt'

property suffixes

A list of the final component's suffixes, if any.

These include the leading periods. For example: ['.tar', '.gz']

with_name(*name*)

Return a new path with the file name changed.

with_stem(*stem*)

Return a new path with the stem changed.

with_suffix(*suffix*)

Return a new path with the file suffix changed. If the path has no suffix, add given suffix. If the given suffix is an empty string, remove the suffix from the path.

AFL.automation.instrument.SeabreezeUVVis.SeabreezeUVVis

class AFL.automation.instrument.SeabreezeUVVis.**SeabreezeUVVis**(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

Bases: *Driver*

__init__(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

Methods

__init__([backend, device_serial, overrides])
collect(nframes[, reduced, absorbance, ...])
collectContinuous(duration[, start, return_data])

continues on next page

Table 8 – continued from previous page

collectSingleSpectrum([set_reference, set_air])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
getExposure()	
getExposureDelay()	
getFilename()	
getFilepath()	
getSaveSingleScan()	
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reduced(data_raw_mean, data_raw_std[, ...])	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
setExposure(time)	
setExposureDelay(time)	
setFilename(filename)	
setFilepath(filepath)	
setSaveSingleScan(saveSingleScan)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	

continues on next page

Table 8 – continued from previous page

set_sample(sample_name[, sample_uuid])
status()
unqueued()

Attributes

defaults

defaults = {'air_uuid': '', 'correctDarkCounts': False, 'correctNonlinearity': False, 'exposure': 0.01, 'exposure_delay': 0, 'filename': 'test.h5', 'filepath': '.', 'reference_uuid': '', 'saveSingleScan': False}

__init__(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

getExposure()

getExposureDelay()

getFilename()

getSaveSingleScan()

getFilepath()

setFilepath(*filepath*)

setFilename(*filename*)

setSaveSingleScan(*saveSingleScan*)

setExposureDelay(*time*)

setExposure(*time*)

collectContinuous(*duration*, *start=None*, *return_data=False*, ***kwargs*)

collectSingleSpectrum(*set_reference=False*, *set_air=False*, ***kwargs*)

collect(*nframes: int*, *reduced: bool = False*, *absorbance: bool = True*, *set_reference: bool = False*, *set_air: bool = False*, *exposure: float | None = None*, *return_data: bool = False*, ***kwargs*)

reduced(*data_raw_mean*, *data_raw_std*, *absorbance=True*, *reference_uuid='reference_uuid'*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

status()

unqueued()

class AFL.automation.instrument.SeabreezeUVVis.**SeabreezeUVVis**(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

defaults = {'air_uuid': '', 'correctDarkCounts': False, 'correctNonlinearity': False, 'exposure': 0.01, 'exposure_delay': 0, 'filename': 'test.h5', 'filepath': '.', 'reference_uuid': '', 'saveSingleScan': False}

__init__(*backend='cseabreeze'*, *device_serial=None*, *overrides=None*)

getExposure()

getExposureDelay()

getFilename()

getSaveSingleScan()

getFilepath()

setFilepath(*filepath*)

setFilename(*filename*)

setSaveSingleScan(*saveSingleScan*)

setExposureDelay(*time*)

setExposure(*time*)

collectContinuous(*duration*, *start=None*, *return_data=False*, ***kwargs*)

collectSingleSpectrum(*set_reference=False*, *set_air=False*, ***kwargs*)

collect(*nframes: int*, *reduced: bool = False*, *absorbance: bool = True*, *set_reference: bool = False*, *set_air: bool = False*, *exposure: float | None = None*, *return_data: bool = False*, ***kwargs*)

reduced(*data_raw_mean*, *data_raw_std*, *absorbance=True*, *reference_uuid='reference_uuid'*)

AFL.automation.instrument.SpecScreen_Driver

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

AFL.automation.instrument.SpecScreen_Driver.ceil

AFL.automation.instrument.SpecScreen_Driver.**ceil**(*x*, */*)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

AFL.automation.instrument.SpecScreen_Driver.listify

AFL.automation.instrument.SpecScreen_Driver.**listify**(*obj*)

AFL.automation.instrument.SpecScreen_Driver.sqrt

AFL.automation.instrument.SpecScreen_Driver.**sqrt**(*x*, /)

Return the square root of x.

Classes

Driver(name[, defaults, overrides])	
SpecScreen_Driver([log_file])	

AFL.automation.instrument.SpecScreen_Driver.Driver

class AFL.automation.instrument.SpecScreen_Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.instrument.SpecScreen_Driver.SpecScreen_Driver

class AFL.automation.instrument.SpecScreen_Driver.**SpecScreen_Driver**(*log_file=None*)

Bases: *Driver*

__init__(*log_file=None*)

Methods

__init__([log_file])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

__init__(*log_file=None*)

status()

execute(***kwargs*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.instrument.SpecScreen_Driver.**SpecScreen_Driver**(*log_file=None*)

__init__(*log_file=None*)

status()

execute(***kwargs*)

6.2.3 Loading

The Loading module provides functionality for loading and handling samples.

AFL.automation.loading

AFL.automation.loading

Modules

CetoniMultiPosValve	
ChemyxSyringePump	This is largely duplicated from the reference code provided by Chemyx.
DigitalOutPressureController	
DoubleViciMultiposSelector	
DummyPump	
FlowSelector	
LoadStopperDriver	
MultiChannelRelay	
NE1kSyringePump	
OneSelectorBlowoutSampleCell	
PneumaticPressureSampleCell	
PneumaticSampleCell	
PressureController	
PressureControllerAsPump	
PushPullSelectorSampleCell	
RSoXSSolutionSampleCell	
SainSmartRelay	
SampleCell	
Sensor	
SensorCallbackThread	
SensorPollingThread	
SerialDevice	
SyringePump	
Tubing	
TwoSelectorBlowoutSampleCell	
UltimusVPressureController	
ViciMultiposSelector	

AFL.automation.loading.CetoniMultiPosValve

Classes

CetoniMultiPosValve(parentpump[, portlabels])	
FlowSelector()	

AFL.automation.loading.CetoniMultiPosValve.CetoniMultiPosValve

class AFL.automation.loading.CetoniMultiPosValve.**CetoniMultiPosValve**(*parentpump*, *portlabels={}*)

Bases: *FlowSelector*

__init__(*parentpump*, *portlabels={}*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

Methods

__init__(parentpump[, portlabels])	connect to valve and query the number of positions
getPort([as_str])	query the current selected position
selectPort(port[, direction])	moves the selector to portnum

__init__(*parentpump*, *portlabels={}*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.CetoniMultiPosValve.FlowSelector

class AFL.automation.loading.CetoniMultiPosValve.**FlowSelector**

Bases: object

__init__()

Methods

__init__()
getPort()
selectPort()

getPort()

selectPort()

class AFL.automation.loading.CetoniMultiPosValve.**CetoniMultiPosValve**(*parentpump*, *portlabels={}*)

__init__(*parentpump*, *portlabels={}*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.ChemyxSyringePump

This is largely duplicated from the reference code provided by Chemyx. Their package is a GUI and direct import of the module would be problematic.

Functions

getOpenPorts()	
parsePortName(portinfo)	On macOS and Linux, selects only usbserial options and parses the 8 character serial number.

AFL.automation.loading.ChemyxSyringePump.getOpenPorts

AFL.automation.loading.ChemyxSyringePump.**getOpenPorts**()

AFL.automation.loading.ChemyxSyringePump.parsePortName

AFL.automation.loading.ChemyxSyringePump.**parsePortName**(*portinfo*)

On macOS and Linux, selects only usbserial options and parses the 8 character serial number.

Classes

ChemyxConnection(port, baudrate[, x, mode, ...])	
ChemyxSyringePump(port, syringe_id_mm, ...)	
SyringePump()	

AFL.automation.loading.ChemyxSyringePump.ChemyxConnection

class AFL.automation.loading.ChemyxSyringePump.**ChemyxConnection**(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

Bases: object

__init__(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

Methods

__init__(port, baudrate[, x, mode, verbose])
addMode(command)
addX(command)
closeConnection()
getDisplacedVolume()
getElapsedTime()
getParameterLimits()
getParameters()
getPumpStatus()
getResponse()
openConnection()
pausePump()
restartPump()
sendCommand(command)
setDelay(delay)
setDiameter(diameter)
setRate(rate)
setTime(timer)
setUnits(units)
setVolume(volume)
startPump()
stopPump()

__init__(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

openConnection()

closeConnection()

sendCommand(*command*)

getResponse()

startPump()

stopPump()

pausePump()

restartPump()

setUnits(*units*)

setDiameter(*diameter*)

setRate(*rate*)

setVolume(*volume*)

setDelay(*delay*)

setTime(*timer*)

getParameterLimits()

getParameters()

getDisplacedVolume()

getElapsedTime()

getPumpStatus()

addMode(*command*)

addX(*command*)

AFL.automation.loading.ChemyxSyringePump.ChemyxSyringePump

class AFL.automation.loading.ChemyxSyringePump.**ChemyxSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

Bases: *SyringePump*

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

Initializes and verifies connection to a Chemyx syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

Methods

__init__(port, syringe_id_mm, syringe_volume)	Initializes and verifies connection to a Chemyx syringe pump.
blockUntilStatusStopped([pollingdelay])	This is a deprecated function from old serial logic.
dispense(volume[, block, delay])	
emptySyringe()	
getLevel()	
getRate()	
getStatus()	query the pump status and return whether the pump is moving or not (true if moving, false if stopped)
getValueFromParams(search_key)	
setLevel(level)	
setRate(rate)	
stop()	Abort the current dispense/withdraw action.
wait_dosage_finished([timeout_seconds])	The function waits until the last dosage command has finished until the timeout occurs.
withdraw(volume[, block, delay])	

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

Initializes and verifies connection to a Chemyx syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

wait_dosage_finished(*timeout_seconds=60*)

The function waits until the last dosage command has finished until the timeout occurs.

stop()

Abort the current dispense/withdraw action.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

getLevel()

setLevel(*level*)

blockUntilStatusStopped(*pollingdelay=0.2*)

This is a deprecated function from old serial logic. It should work, but do not use.

getStatus()

query the pump status and return whether the pump is moving or not (true if moving, false if stopped)

getValueFromParams(*search_key*)

AFL.automation.loading.ChemyxSyringePump.SyringePump

class AFL.automation.loading.ChemyxSyringePump.**SyringePump**

Bases: object

__init__()

Methods

__init__()
dispense(volume[, block])
emptySyringe()
getRate(rate)
setRate(rate)
stop()
withdraw(volume[, block])

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

class AFL.automation.loading.ChemyxSyringePump.**ChemyxSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *flow_delay=5*)

Initializes and verifies connection to a Chemyx syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

wait_dosage_finished(*timeout_seconds=60*)

The function waits until the last dosage command has finished until the timeout occurs.

stop()

Abort the current dispense/withdraw action.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

getLevel()

setLevel(*level*)

blockUntilStatusStopped(*pollingdelay=0.2*)

This is a deprecated function from old serial logic. It should work, but do not use.

getStatus()

query the pump status and return whether the pump is moving or not (true if moving, false if stopped)

getValueFromParams(*search_key*)

AFL.automation.loading.ChemyxSyringePump.**getOpenPorts**()

AFL.automation.loading.ChemyxSyringePump.**parsePortName**(*portinfo*)

On macOS and Linux, selects only usbserial options and parses the 8 character serial number.

class AFL.automation.loading.ChemyxSyringePump.**ChemyxConnection**(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

__init__(*port*, *baudrate*, *x=0*, *mode=0*, *verbose=False*)

openConnection()

closeConnection()

sendCommand(*command*)

getResponse()

startPump()

stopPump()

pausePump()

restartPump()

setUnits(*units*)

setDiameter(*diameter*)

setRate(*rate*)

setVolume(*volume*)

setDelay(*delay*)

setTime(*timer*)

getParameterLimits()

getParameters()

getDisplacedVolume()

getElapsedTime()

getPumpStatus()

addMode(*command*)

addX(*command*)

AFL.automation.loading.DigitalOutPressureController

Classes

DigitalOutPressureController(digital_out, ...)	
PressureController()	Abstract superclass for pressure controllers that provides timed dispensing

AFL.automation.loading.DigitalOutPressureController.DigitalOutPressureController

class AFL.automation.loading.DigitalOutPressureController.**DigitalOutPressureController**(*digital_out*, *pressure_to_v_conv*)

Bases: *PressureController*

__init__(*digital_out*, *pressure_to_v_conv*)

Initializes a DigitalOutPressureController

Params:

digital_out (AFL.automation.DigitalOut): pressure_to_v_conv (float): pressure units per volt

Methods

__init__(digital_out, pressure_to_v_conv)	Initializes a DigitalOutPressureController
blockUntilStatusStopped([pollingdelay])	block execution until the controller finishes a dispense
dispenseRunning()	Returns true if a timed dispense is running, false otherwise.
ramp_dispense(dispense_start_pressure, ...)	Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*.
set_P(pressure)	
stop()	Abort the current timed dispense action.
timed_dispense(dispense_pressure, dispense_time)	Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*.

__init__(*digital_out*, *pressure_to_v_conv*)

Initializes a DigitalOutPressureController

Params:

digital_out (AFL.automation.DigitalOut): pressure_to_v_conv (float): pressure units per volt

set_P(*pressure*)

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

AFL.automation.loading.DigitalOutPressureController.PressureController

class AFL.automation.loading.DigitalOutPressureController.**PressureController**

Bases: object

Abstract superclass for pressure controllers that provides timed dispensing

__init__()

Methods

__init__()	
blockUntilStatusStopped([pollingdelay])	block execution until the controller finishes a dispense
dispenseRunning()	Returns true if a timed dispense is running, false otherwise.
ramp_dispense(dispense_start_pressure, ...)	Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*.
stop()	Abort the current timed dispense action.
timed_dispense(dispense_pressure, dispense_time)	Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*.

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

class AFL.automation.loading.DigitalOutPressureController.**DigitalOutPressureController**(*digital_out*, *pressure_to_v_conv*)

__init__(*digital_out*, *pressure_to_v_conv*)

Initializes a DigitalOutPressureController

Params:

digital_out (AFL.automation.DigitalOut): pressure_to_v_conv (float): pressure units per volt

set_P(*pressure*)

AFL.automation.loading.DoubleViciMultiposSelector

Classes

DoubleViciMultiposSelector(port1, port2[, ...])	
FlowSelector()	
ViciMultiposSelector(port[, baudrate, ...])	

AFL.automation.loading.DoubleViciMultiposSelector.DoubleViciMultiposSelector

class AFL.automation.loading.DoubleViciMultiposSelector.**DoubleViciMultiposSelector**(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

Bases: *FlowSelector*

__init__(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

Methods

__init__(port1, port2[, baudrate, portlabels])	connect to valve and query the number of positions
getPort([as_str])	query the current selected position
selectPort(port[, direction])	moves the selector to portnum

__init__(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.DoubleViciMultiposSelector.FlowSelector

class AFL.automation.loading.DoubleViciMultiposSelector.**FlowSelector**

Bases: object

__init__()

Methods

__init__()
getPort()
selectPort()

getPort()

selectPort()

AFL.automation.loading.DoubleViciMultiposSelector.ViciMultiposSelector

class AFL.automation.loading.DoubleViciMultiposSelector.**ViciMultiposSelector**(*port*, *baudrate=9600*, *portlabels=None*)

Bases: *SerialDevice*, *FlowSelector*

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

Methods

__init__(port[, baudrate, portlabels])	connect to valve and query the number of positions
getPort([as_str])	query the current selected position
selectPort(port[, direction, block])	moves the selector to portnum
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*, *block=True*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

Parameters

- **port** (*String or int*) – the name, or number, of the port to switch to
- **direction** (*String, default=None*) – direction "CW" or "CCW" to perform the move in
- **block** (*bool, default=True*) – block return until move completes

getPort(*as_str=False*)

query the current selected position

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

class AFL.automation.loading.DoubleViciMultiposSelector.**DoubleViciMultiposSelector**(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

__init__(*port1*, *port2*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)

- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

getPort(*as_str=False*)

query the current selected position

AFL.automation.loading.DummyPump

Classes

DummyPump()	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

AFL.automation.loading.DummyPump.DummyPump

class AFL.automation.loading.DummyPump.**DummyPump**

Bases: *SyringePump*

__init__()

Dummy pump for testing - does nothing, but boy does it look good doing it.

Methods

__init__()	Dummy pump for testing - does nothing, but boy does it look good doing it.
blockUntilStatusStopped([pollingdelay])	
dispense(volume[, block, delay])	
emptySyringe()	
getRate()	
getStatus()	query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)
setRate(rate)	
stop()	Abort the current dispense/withdraw action.
withdraw(volume[, block, delay])	

__init__()

Dummy pump for testing - does nothing, but boy does it look good doing it.

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.DummyPump.SerialDevice

class AFL.automation.loading.DummyPump.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.DummyPump.SyringePump

class AFL.automation.loading.DummyPump.**SyringePump**

Bases: object

__init__()

Methods

__init__()	
dispense(volume[, block])	
emptySyringe()	
getRate(rate)	
setRate(rate)	
stop()	
withdraw(volume[, block])	

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

class AFL.automation.loading.DummyPump.**DummyPump**

__init__()

Dummy pump for testing - does nothing, but boy does it look good doing it.

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.FlowSelector

Classes

FlowSelector()	

AFL.automation.loading.FlowSelector.FlowSelector

class AFL.automation.loading.FlowSelector.**FlowSelector**

Bases: object

__init__()

Methods

__init__()	
getPort()	
selectPort()	

getPort()

selectPort()

class AFL.automation.loading.FlowSelector.**FlowSelector**

getPort()

selectPort()

AFL.automation.loading.LoadStopperDriver

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
Driver(name[, defaults, overrides])	
LoadStopperDriver(sensor[, load_client, ...])	Driver for stopping loads
SensorPollingThread(sensor[, period, ...])	
StopLoadCBv1(poll, period, load_client[, ...])	
StopLoadCBv2(poll, period[, load_client, ...])	

AFL.automation.loading.LoadStopperDriver.Client

class AFL.automation.loading.LoadStopperDriver.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	

continues on next page

Table 9 – continued from previous page

logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.loading.LoadStopperDriver.Driver

class AFL.automation.loading.LoadStopperDriver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.LoadStopperDriver.LoadStopperDriver

class AFL.automation.loading.LoadStopperDriver.**LoadStopperDriver**(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

Bases: *Driver*

Driver for stopping loads

__init__(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

Methods

__init__(sensor[, load_client, load_object, ...])	
calibrate_sensor()	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
read_poll()	
read_poll_load()	
read_sensor()	
reset()	
reset_poll()	
reset_sample()	
reset_stopper()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

Attributes

app
defaults

defaults = {'load_speed': 2, 'period': 0.05, 'poll_window': 1000, 'sensorlabel': '', 'stopper_baseline_duration': 2, 'stopper_filepath': '/github/home/.afl/loadstopper_data', 'stopper_loadstop_cooldown': 2, 'stopper_min_load_time': 3, 'stopper_post_detection_sleep': 1, 'stopper_threshold_npts': 50, 'stopper_threshold_std': 3.0, 'stopper_threshold_v_step': 1, 'stopper_timeout': 120}

__init__(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

status()

property app

calibrate_sensor()

read_sensor()

read_poll()

read_poll_load()

reset()

reset_poll()

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

reset_stopper()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

AFL.automation.loading.LoadStopperDriver.SensorPollingThread

class AFL.automation.loading.LoadStopperDriver.**SensorPollingThread**(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

Bases: Thread

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(sensor[, period, callback, ...])	This constructor should always be called with keyword arguments.
alive()	
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
read()	
read_load_buffer()	
reset_load_buffer()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

read_load_buffer()

reset_load_buffer()

terminate()

alive()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call

is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

AFL.automation.loading.LoadStopperDriver.StopLoadCBv1

class AFL.automation.loading.LoadStopperDriver.**StopLoadCBv1**(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

Bases: *SensorCallbackThread*

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll, period, load_client[, ...])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

terminate()

update_status(*value*)

AFL.automation.loading.LoadStopperDriver.StopLoadCBv2

class AFL.automation.loading.LoadStopperDriver.**StopLoadCBv2**(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

Bases: *SensorCallbackThread*

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll, period[, load_client, ...])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

terminate()

update_status(*value*)

class AFL.automation.loading.LoadStopperDriver.**LoadStopperDriver**(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

Driver for stopping loads

```
defaults = {'load_speed': 2, 'period': 0.05, 'poll_window': 1000, 'sensorlabel':
'', 'stopper_baseline_duration': 2, 'stopper_filepath':
'/github/home/.afl/loadstopper_data', 'stopper_loadstop_cooldown': 2,
'stopper_min_load_time': 3, 'stopper_post_detection_sleep': 1,
'stopper_threshold_npts': 50, 'stopper_threshold_std': 3.0,
'stopper_threshold_v_step': 1, 'stopper_timeout': 120}
```

__init__(*sensor*, *load_client=None*, *load_object=None*, *auto_initialize=True*, *overrides=None*, *data=None*, *sensorlabel=''*, *name='LoadStopperDriver'*)

status()

property app

calibrate_sensor()

read_sensor()

read_poll()

read_poll_load()

reset()

reset_poll()

reset_stopper()

AFL.automation.loading.MultiChannelRelay

Classes

MultiChannelRelay()

AFL.automation.loading.MultiChannelRelay.MultiChannelRelay

class AFL.automation.loading.MultiChannelRelay.**MultiChannelRelay**

Bases: object

__init__()

Methods

__init__()
getChannels(channels)
setChannels(channels)
toggleChannels(channels)

setChannels(*channels*)

getChannels(*channels*)

toggleChannels(*channels*)

class AFL.automation.loading.MultiChannelRelay.**MultiChannelRelay**

setChannels(*channels*)

getChannels(*channels*)

toggleChannels(*channels*)

AFL.automation.loading.NE1kSyringePump

Classes

NE1kSyringePump(port, syringe_id_mm, ...[, ...])	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

AFL.automation.loading.NE1kSyringePump.NE1kSyringePump

class AFL.automation.loading.NE1kSyringePump.**NE1kSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

Bases: *SyringePump*

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

Initializes and verifies connection to a New Era 1000 syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

daisy_chain = used for the 'party-line' mode on these pumps where a string of pumps is on one serial port.

when setting up daisy chaining:
connect to the first pump on a port with daisy_chain = False on subsequent pumps, set daisy_chain to the pump with a hardware connection (the first pump)

or any other pump on the string.

note: when daisy chaining you should probably set pumpid explicitly rather than autodiscovering
as most likely the autodiscovery will return the first pump id each time.

pumpid = the ID configured in the pump firmware. If not set, will attempt to auto-discover a pump.
setting pumpid will save some time on connection and probably result in more reproducible behavior. Practically mandatory for daisy chain mode.

Methods

__init__(port, syringe_id_mm, syringe_volume)	Initializes and verifies connection to a New Era 1000 syringe pump.
blockUntilStatusStopped([pollingdelay])	
dispense(volume[, block, delay])	
emptySyringe()	
getRate()	
getStatus()	query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)
setRate(rate)	
stop()	Abort the current dispense/withdraw action.
withdraw(volume[, block, delay])	

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

Initializes and verifies connection to a New Era 1000 syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

daisy_chain = used for the 'party-line' mode on these pumps where a string of pumps is on one serial port.

when setting up daisy chaining:
connect to the first pump on a port with daisy_chain = False on subsequent pumps, set daisy_chain to the pump with a hardware connection (the first pump)

or any other pump on the string.

note: when daisy chaining you should probably set pumpid explicitly rather than autodiscovering
as most likely the autodiscovery will return the first pump id each time.

pumpid = the ID configured in the pump firmware. If not set, will attempt to auto-discover a pump.
setting pumpid will save some time on connection and probably result in more reproducible behavior. Practically mandatory for daisy chain mode.

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.NE1kSyringePump.SerialDevice

class AFL.automation.loading.NE1kSyringePump.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.NE1kSyringePump.SyringePump

class AFL.automation.loading.NE1kSyringePump.**SyringePump**

Bases: object

__init__()

Methods

__init__()	
dispense(volume[, block])	
emptySyringe()	
getRate(rate)	
setRate(rate)	
stop()	
withdraw(volume[, block])	

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

class AFL.automation.loading.NE1kSyringePump.**NE1kSyringePump**(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

__init__(*port*, *syringe_id_mm*, *syringe_volume*, *baud=9600*, *daisy_chain=None*, *pumpid=None*, *flow_delay=5*)

Initializes and verifies connection to a New Era 1000 syringe pump.

port = serial port reference

syringe_id_mm = syringe inner diameter in mm, used for absolute volume.
(will re-program the pump with this diameter on connection)

syringe_volume = syringe volume in mL

baud = baudrate for connection

daisy_chain = used for the 'party-line' mode on these pumps where a string of pumps is on one serial port.

when setting up daisy chaining:
connect to the first pump on a port with daisy_chain = False on subsequent pumps, set daisy_chain to the pump with a hardware connection (the first pump)

or any other pump on the string.

note: when daisy chaining you should probably set pumpid explicitly rather than autodiscovering
as most likely the autodiscovery will return the first pump id each time.

pumpid = the ID configured in the pump firmware. If not set, will attempt to auto-discover a pump.
setting pumpid will save some time on connection and probably result in more reproducible behavior. Practically mandatory for daisy chain mode.

stop()
Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.OneSelectorBlowoutSampleCell

Classes

Driver(name[, defaults, overrides])	
OneSelectorBlowoutSampleCell(pump, selector)	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
TwoSelectorBlowoutSampleCell(pump, selector, ...)	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.OneSelectorBlowoutSampleCell.Driver

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.OneSelectorBlowoutSampleCell.OneSelectorBlowoutSampleCell

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**OneSelectorBlowoutSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level*, *overrides=None*)

Bases: *TwoSelectorBlowoutSampleCell*

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, selector[, rinse_tank_level, ...])	ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells
blowOutCell([cellname, waittime])	
blowOutCellLegacy([cellname])	
catchToSyringe([sampleVolume])	
catchToWaste([sampleVolume])	
cellToWaste([cellname])	
deposit_obj(obj[, uid])	Store an object in the dropbox
drySyringe([blow, waittime])	transfer from air to waste, to push out any residual liquid.
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	

continues on next page

Table 10 – continued from previous page

loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
reset_tank_levels([rinse, waste, cell_waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll([cellname])	
rinseCatch()	
rinseCell([cellname])	
rinseCellFlood([cellname])	
rinseCellPull([cellname])	
rinseSyringe()	
setRinseLevel(vol)	
setWasteLevel(vol)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
swish(vol)	
transfer(source, dest, vol_source[, vol_dest])	
unqueued()	

Attributes

app
defaults

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

property app

catchToSyringe(*sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

loadSample(*cellname='cell'*, *sampleVolume=0*)

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

rinseAll(*cellname='cell'*)

rinseCatch()

rinseCell(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseSyringe()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

status()

swish(*vol*)

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

unqueued()

AFL.automation.loading.OneSelectorBlowoutSampleCell.TwoSelectorBlowoutSampleCell

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**TwoSelectorBlowoutSampleCell**(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, selector, blowselector[, ...])	ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells
blowOutCell([cellname, waittime])	

continues on next page

Table 11 – continued from previous page

blowOutCellLegacy([cellname])	
catchToSyringe([sampleVolume])	
catchToWaste([sampleVolume])	
cellToWaste([cellname])	
deposit_obj(obj[, uid])	Store an object in the dropbox
drySyringe([blow, waittime])	transfer from air to waste, to push out any residual liquid.
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
reset_tank_levels([rinse, waste, cell_waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll([cellname])	
rinseCatch()	
rinseCell([cellname])	
rinseCellFlood([cellname])	
rinseCellPull([cellname])	
rinseSyringe()	
setRinseLevel(vol)	
setWasteLevel(vol)	

continues on next page

Table 11 – continued from previous page

set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
swish(vol)	
transfer(source, dest, vol_source[, vol_dest])	
unqueued()	

Attributes

app
defaults

defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1, 'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2, 'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol': 1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0, 'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2, 'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3, 'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3, 'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol': 1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

setWasteLevel(*vol*)

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

AFL.automation.loading.OneSelectorBlowoutSampleCell.defaultdict

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.OneSelectorBlowoutSampleCell.**OneSelectorBlowoutSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

AFL.automation.loading.PneumaticPressureSampleCell

Classes

Driver(name[, defaults, overrides])	
PneumaticPressureSampleCell(pctrl, relayboard)	Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.
SampleCell()	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.PneumaticPressureSampleCell.Driver

class AFL.automation.loading.PneumaticPressureSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.PneumaticPressureSampleCell.PneumaticPressureSampleCell

class AFL.automation.loading.PneumaticPressureSampleCell.**PneumaticPressureSampleCell**(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=N*, *overrides=None*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a variable-pressure regulator.

__init__(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=None*, *overrides=None*)

pctrl: a pressurecontroller object supporting the set_P method() and the optional p_ramp() method.
e.g. pctrl = DigitalOutPressureController(LabJackDigitalOut(...))

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pctrl, relayboard[, digitalin, ...])	pctrl: a pressurecontroller object supporting the set_P method() and the optional p_ramp() method.
advanceSample([load_dest_label])	Move a sample from one measurement cell to the next
calibrate_sensor()	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
get_sensor_config(**kwargs)	
loadSample([cellname, sampleVolume, ...])	Load a sample into the cell
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute

continues on next page

Table 12 – continued from previous page

primeRinse([waittime])	
queued()	
quickbar()	
read_sensor()	
read_sensor_poll(**kwargs)	
read_sensor_poll_load(**kwargs)	
reset_sample()	
reset_tank_levels([rinse1, rinse2, waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll()	
rinseCell([cellname])	
sensor_reset([sensor_n])	
setRinseLevel(vol)	
setWasteLevel(vol)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
set_sensor_config(**kwargs)	
status()	
stopLoad(**kwargs)	
unqueued()	

Attributes

app
data
defaults

```
defaults = {'arm_move_delay': 0.2, 'blowout_pressure': 20,
'external_load_complete_trigger': False, 'load_mode': 'static', 'load_pressure':
2, 'load_timeout': 60, 'ramp_load_duration': 20, 'ramp_load_stop_pressure': 7,
'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None,
0.5), ('blow', 5)], 'vent_delay': 0.5}
```

__init__(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=None*, *overrides=None*)

pctrl: a pressurecontroller object supporting the set_P method() and the optional p_ramp() method.
e.g. pctrl = DigitalOutPressureController(LabJackDigitalOut(...))

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

property data

status()

loadSample(*cellname='cell'*, *sampleVolume=None*, *load_dest_label=''*)

Load a sample into the cell

Params *cellname* and *sampleVolume* are kept for backward compat, but are deprecated and unused.

advanceSample(*load_dest_label=''*)

Move a sample from one measurement cell to the next

Params:

load_dest_label (str, default ''): a 'destination label' for this load.
labeled sensors will only stop the load if their name is in this destination label. example:

sensor 1 labeled 'afterSANS' sensor 2 labeled 'beforeSPEC' sensor 3 labeled '' (default)

advanceSample(load_dest_label='afterSANS') –> sensor 1 or sensor 3 can stop it advanceSample(load_dest_label='beforeSPEC afterSANS') –> sensor 1, sensor 2, or sensor 3 can stop it advanceSample(load_dest_label='') –> only sensor 3 can stop it

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

read_sensor()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

read_sensor_poll(***kwargs*)

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset(*sensor_n=None*)

AFL.automation.loading.PneumaticPressureSampleCell.SampleCell

class AFL.automation.loading.PneumaticPressureSampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()
loadSample()

loadSample()

AFL.automation.loading.PneumaticPressureSampleCell.defaultdict

class AFL.automation.loading.PneumaticPressureSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.PneumaticPressureSampleCell.**PneumaticPressureSampleCell**(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=N*
overrides=None)

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a variable-pressure regulator.

defaults = {'arm_move_delay': 0.2, 'blowout_pressure': 20, 'external_load_complete_trigger': False, 'load_mode': 'static', 'load_pressure': 2, 'load_timeout': 60, 'ramp_load_duration': 20, 'ramp_load_stop_pressure': 7, 'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None, 0.5), ('blow', 5)], 'vent_delay': 0.5}

__init__(*pctrl*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *robot_interlock_host=None*, *overrides=None*)

pctrl: a pressurecontroller object supporting the set_P method() and the optional p_ramp() method.
e.g. pctrl = DigitalOutPressureController(LabJackDigitalOut(...))

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

property data

status()

loadSample(*cellname='cell'*, *sampleVolume=None*, *load_dest_label=''*)

Load a sample into the cell

Params *cellname* and *sampleVolume* are kept for backward compat, but are deprecated and unused.

advanceSample(*load_dest_label=''*)

Move a sample from one measurement cell to the next

Params:

load_dest_label (str, default ‘’): a ‘destination label’ for this load.
labeled sensors will only stop the load if their name is in this destination label. example:

sensor 1 labeled ‘afterSANS’ sensor 2 labeled ‘beforeSPEC’ sensor 3 labeled ‘’ (default)

advanceSample(load_dest_label=’afterSANS’) –> sensor 1 or sensor 3 can stop it advanceSample(load_dest_label=’beforeSPEC afterSANS’) –> sensor 1, sensor 2, or sensor 3 can stop it advanceSample(load_dest_label=’’) –> only sensor 3 can stop it

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

read_sensor()

read_sensor_poll(***kwargs*)

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset(*sensor_n=None*)

AFL.automation.loading.PneumaticSampleCell

Classes

Driver(name[, defaults, overrides])	
PneumaticSampleCell(pump, relayboard[, ...])	Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.
SampleCell()	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.PneumaticSampleCell.Driver

class AFL.automation.loading.PneumaticSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.PneumaticSampleCell.PneumaticSampleCell

class AFL.automation.loading.PneumaticSampleCell.**PneumaticSampleCell**(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a syringe pump.

__init__(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, relayboard[, digitalin, ...])	pump: a pump object supporting withdraw() and dispense() methods
calibrate_sensor()	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
get_sensor_config(**kwargs)	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
primeRinse([waittime])	
queued()	

continues on next page

Table 13 – continued from previous page

quickbar()	
read_sensor()	
read_sensor_poll(**kwargs)	
read_sensor_poll_load(**kwargs)	
reset_pump([dispense])	
reset_sample()	
reset_tank_levels([rinse1, rinse2, waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll()	
rinseCell([cellname])	
sensor_reset()	
setRinseLevel(vol)	
setWasteLevel(vol)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
set_sensor_config(**kwargs)	
status()	
stopLoad(**kwargs)	
unqueued()	

Attributes

app	
defaults	

defaults = {'air_speed': 30, 'arm_move_delay': 0.2, 'catch_to_cell_vol': 1.15, 'external_load_complete_trigger': False, 'large_dispense_vol': 5, 'load_speed': 2, 'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None, 0.5), ('blow', 5)], 'vent_delay': 0.5, 'withdraw_vol': 1.5}

__init__(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_pump(*dispense=False*)

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

status()

loadSample(*cellname='cell'*, *sampleVolume=0*)

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

read_sensor()

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

read_sensor_poll(***kwargs*)

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset()

AFL.automation.loading.PneumaticSampleCell.SampleCell

class AFL.automation.loading.PneumaticSampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()	
loadSample()	

loadSample()

AFL.automation.loading.PneumaticSampleCell.defaultdict

class AFL.automation.loading.PneumaticSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.PneumaticSampleCell.**PneumaticSampleCell**(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

Class for a sample cell consisting of a push-through, pneumatically-closed sample loader.

Driven by a syringe pump.

```
defaults = {'air_speed': 30, 'arm_move_delay': 0.2, 'catch_to_cell_vol': 1.15,
'external_load_complete_trigger': False, 'large_dispense_vol': 5, 'load_speed':
2, 'rinse_program': [('rinse1', 5), (None, 2), ('rinse2', 5), ('blow', 5), (None,
0.5), ('blow', 5)], 'vent_delay': 0.5, 'withdraw_vol': 1.5}
```

__init__(*pump*, *relayboard*, *digitalin=None*, *rinse1_tank_level=950*, *rinse2_tank_level=950*, *waste_tank_level=0*, *load_stopper=None*, *overrides=None*)

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

relayboard: a relay board object supporting string-based setChannels() method
required channels are 'arm-up','arm-down', 'rinse1','rinse2','blow','enable','piston-vent','postsample'
e.g. selector = SainSmartRelay(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_pump(*dispense=False*)

reset_tank_levels(*rinse1=950*, *rinse2=950*, *waste=0*)

property app

status()

loadSample(*cellname='cell'*, *sampleVolume=0*)

stopLoad(***kwargs*)

rinseCell(*cellname='cell'*)

rinseAll()

setRinseLevel(*vol*)

setWasteLevel(*vol*)

primeRinse(*waittime=10*)

calibrate_sensor()

read_sensor()

read_sensor_poll(***kwargs*)

read_sensor_poll_load(***kwargs*)

set_sensor_config(***kwargs*)

get_sensor_config(***kwargs*)

sensor_reset()

AFL.automation.loading.PressureController

Classes

PressureController()	Abstract superclass for pressure controllers that provides timed dispensing

AFL.automation.loading.PressureController.PressureController

class AFL.automation.loading.PressureController.**PressureController**

Bases: object

Abstract superclass for pressure controllers that provides timed dispensing

__init__()

Methods

__init__()	
blockUntilStatusStopped([pollingdelay])	block execution until the controller finishes a dispense
dispenseRunning()	Returns true if a timed dispense is running, false otherwise.
ramp_dispense(dispense_start_pressure, ...)	Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*.
stop()	Abort the current timed dispense action.
timed_dispense(dispense_pressure, dispense_time)	Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*.

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

class AFL.automation.loading.PressureController.**PressureController**

Abstract superclass for pressure controllers that provides timed dispensing

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

AFL.automation.loading.PressureControllerAsPump

Classes

PressureControllerAsPump(pressure_controller)	
SerialDevice(port[, baudrate, timeout, ...])	
SyringePump()	

AFL.automation.loading.PressureControllerAsPump.PressureControllerAsPump

class AFL.automation.loading.PressureControllerAsPump.**PressureControllerAsPump**(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

Bases: *SyringePump*

__init__(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

Initialize a pressure controller as a syringe pump.

pressure_controller: controller to connect to dispense_pressure: pressure at which dispense should run implied_flow_rate: flow rate which should be used to convert to dispense times, mL/min

Methods

__init__(pressure_controller[, ...])	Initialize a pressure controller as a syringe pump.
blockUntilStatusStopped([pollingdelay])	
dispense(volume[, block, delay])	
emptySyringe()	
getRate()	
getStatus()	query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)
setRate(rate)	
stop()	Abort the current dispense/withdraw action.
withdraw(volume[, block, delay])	

__init__(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

Initialize a pressure controller as a syringe pump.

pressure_controller: controller to connect to dispense_pressure: pressure at which dispense should run implied_flow_rate: flow rate which should be used to convert to dispense times, mL/min

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.PressureControllerAsPump.SerialDevice

class AFL.automation.loading.PressureControllerAsPump.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.PressureControllerAsPump.SyringePump

class AFL.automation.loading.PressureControllerAsPump.**SyringePump**

Bases: object

__init__()

Methods

__init__()	
dispense(volume[, block])	
emptySyringe()	
getRate(rate)	
setRate(rate)	
stop()	
withdraw(volume[, block])	

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

class AFL.automation.loading.PressureControllerAsPump.**PressureControllerAsPump**(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

__init__(*pressure_controller*, *dispense_pressure=3.5*, *implied_flow_rate=50*)

Initialize a pressure controller as a syringe pump.

pressure_controller: controller to connect to dispense_pressure: pressure at which dispense should run implied_flow_rate: flow rate which should be used to convert to dispense times, mL/min

stop()

Abort the current dispense/withdraw action. Equivalent to pressing stop button on panel.

withdraw(*volume*, *block=True*, *delay=True*)

dispense(*volume*, *block=True*, *delay=True*)

setRate(*rate*)

getRate()

emptySyringe()

blockUntilStatusStopped(*pollingdelay=0.2*)

getStatus()

query the pump status and return a tuple of the status character, infused volume, and withdrawn volume)

AFL.automation.loading.PushPullSelectorSampleCell

Classes

Driver(name[, defaults, overrides])	
PushPullSelectorSampleCell(pump, selector[, ...])	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
SampleCell()	
Tubing(specid, length)	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.PushPullSelectorSampleCell.Driver

class AFL.automation.loading.PushPullSelectorSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.PushPullSelectorSampleCell.PushPullSelectorSampleCell

class AFL.automation.loading.PushPullSelectorSampleCell.**PushPullSelectorSampleCell**(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_v*, *calibrated_syringe_to_cell_vo*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector) @TODO: figure out when/where to pull in air to the syringe to make up for extra_vol_to_empty_ml

__init__(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_vol=None*, *calibrated_syringe_to_cell_vol=None*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, selector[, ncells, ...])	ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells
blowOutCell([cellname])	

continues on next page

Table 14 – continued from previous page

blowOutCellForcedAir([cellname, waittime])	
catchToSyringe([sampleVolume])	
catchToWaste([sampleVolume])	
cellToWaste([cellname])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
getParameters()	
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll([cellname])	
rinseCatch()	
rinseCell([cellname])	
rinseCellFlood([cellname])	
rinseCellPull([cellname])	
rinseSyringe()	
setParameter(parameter, value)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object

continues on next page

Table 14 – continued from previous page

set_object([serialized])
set_sample(sample_name[, sample_uuid])
status()
swish(vol)
transfer(source, dest, vol_source[, vol_dest])
unqueued()

Attributes

app

__init__(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_vol=None*, *calibrated_syringe_to_cell_vol=None*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

property app

status()

setParameter(*parameter*, *value*)

getParameters()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

swish(*vol*)

unqueued()

blowOutCell(*cellname='cell'*)

blowOutCellForcedAir(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

AFL.automation.loading.PushPullSelectorSampleCell.SampleCell

class AFL.automation.loading.PushPullSelectorSampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()	
loadSample()	

loadSample()

AFL.automation.loading.PushPullSelectorSampleCell.Tubing

class AFL.automation.loading.PushPullSelectorSampleCell.**Tubing**(*specid*, *length*)

Bases: object

__init__(*specid*, *length*)

length is in cm?

Methods

__init__(specid, length)	length is in cm?
volume()	returns volume in mL

Attributes

tubing

```
tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material':
'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075,
'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in':
0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in':
0.075, 'material': 'Tefzel', 'typeid': 1517}]
```

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

AFL.automation.loading.PushPullSelectorSampleCell.defaultdict

class AFL.automation.loading.PushPullSelectorSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.PushPullSelectorSampleCell.**PushPullSelectorSampleCell**(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_v*, *calibrated_syringe_to_cell_vo*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector) @TODO: figure out when/where to pull in air to the syringe to make up for extra_vol_to_empty_ml

__init__(*pump*, *selector*, *ncells=1*, *thickness=None*, *catch_to_sel_vol=None*, *cell_to_sel_vol=None*, *syringe_to_sel_vol=None*, *selector_internal_vol=None*, *calibrated_catch_to_syringe_vol=None*, *calibrated_syringe_to_cell_vol=None*, *rinse_speed=50.0*, *load_speed=10.0*, *rinse_flow_delay=3.0*, *load_flow_delay=10.0*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

property app

status()

setParameter(*parameter*, *value*)

getParameters()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCell(*cellname='cell'*)

blowOutCellForcedAir(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

AFL.automation.loading.RSoXSSolutionSampleCell

Classes

Driver(name[, defaults, overrides])	
RSoXSSolutionSampleCell(pump, selector[, ...])	Class for a sample cell consisting of a pump and a one-to-many flow selector for a flowrate-limited measurement cell (e.g., a liquid TEM sample holder for RSoXS).
SampleCell()	
Tubing(specid, length)	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.RSoXSSolutionSampleCell.Driver

class AFL.automation.loading.RSoXSSolutionSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.RSoXSSolutionSampleCell.RSoXSSolutionSampleCell

class AFL.automation.loading.RSoXSSolutionSampleCell.**RSoXSSolutionSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a pump and a one-to-many flow selector for a flowrate-limited measurement cell (e.g., a liquid TEM sample holder for RSoXS).

The pump draws from any of the sample ports and can quickly purge/rinse itself and the tubing between the loader and the cell. When the pump is connected to the cell, its max rate is limited to very slow (5 ul/min for RSoXS).

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, selector[, rinse_tank_level, ...])	ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells
blowOutCell([cellname, waittime])	
blowOutCellLegacy([cellname])	
catchToSyringe([sampleVolume])	
catchToWaste([sampleVolume])	
cellToWaste([cellname])	
deposit_obj(obj[, uid])	Store an object in the dropbox
drySyringe([blow, waittime])	transfer from air to waste, to push out any residual liquid.
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	

continues on next page

Table 15 – continued from previous page

get_sample()	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
reset_tank_levels([rinse, waste, cell_waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll([cellname])	
rinseCatch()	
rinseCell([cellname])	
rinseCellFlood([cellname])	
rinseCellPull([cellname])	
rinseSyringe()	
setRinseLevel(vol)	
setWasteLevel(vol)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
swish(vol)	
transfer(source, dest, vol_source[, vol_dest])	
unqueued()	

Attributes

app
defaults

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters

uid (*str*) – The uuid of the file to retrieve

setWasteLevel(*vol*)

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

AFL.automation.loading.RSoXSSolutionSampleCell.SampleCell

class AFL.automation.loading.RSoXSSolutionSampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()	
loadSample()	

loadSample()

AFL.automation.loading.RSoXSSolutionSampleCell.Tubing

class AFL.automation.loading.RSoXSSolutionSampleCell.**Tubing**(*specid*, *length*)

Bases: object

__init__(*specid*, *length*)

length is in cm?

Methods

__init__(specid, length)	length is in cm?
volume()	returns volume in mL

Attributes

tubing

tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material': 'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in': 0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1517}]

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

AFL.automation.loading.RSoXSSolutionSampleCell.defaultdict

class AFL.automation.loading.RSoXSSolutionSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, /)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.RSoXSSolutionSampleCell.**RSoXSSolutionSampleCell**(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector for a flowrate-limited measurement cell (e.g., a liquid TEM sample holder for RSoXS).

The pump draws from any of the sample ports and can quickly purge/rinse itself and the tubing between the loader and the cell. When the pump is connected to the cell, its max rate is limited to very slow (5 ul/min for RSoXS).

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

__init__(*pump*, *selector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

setWasteLevel(*vol*)

AFL.automation.loading.SainSmartRelay

Module Attributes

usbrelay

AFL.automation.loading.SainSmartRelay.usbrelay

AFL.automation.loading.SainSmartRelay.**usbrelay = [[b':FE0100200000FF\r\n', b':FE0100000010F1\r\n'], [b':FE0500000000FD\r\n', b':FE050000FF00FE\r\n'], [b':FE0500010000FC\r\n', b':FE050001FF00FD\r\n'], [b':FE0500020000FB\r\n', b':FE050002FF00FC\r\n'], [b':FE0500030000FA\r\n', b':FE050003FF00FB\r\n'], [b':FE0500040000F9\r\n', b':FE050004FF00FA\r\n'], [b':FE0500050000F8\r\n', b':FE050005FF00F9\r\n'], [b':FE0500060000F7\r\n', b':FE050006FF00F8\r\n'], [b':FE0500070000F6\r\n', b':FE050007FF00F7\r\n'], [b':FE0500080000F5\r\n', b':FE050008FF00F6\r\n'], [b':FE0500090000F4\r\n', b':FE050009FF00F5\r\n'], [b':FE05000A0000F3\r\n', b':FE05000AFF00F4\r\n'], [b':FE05000B0000F2\r\n', b':FE05000BFF00F3\r\n'], [b':FE05000C0000F1\r\n', b':FE05000CFF00F2\r\n'], [b':FE05000D0000F0\r\n', b':FE05000DFF00F1\r\n'], [b':FE05000E0000FF\r\n', b':FE05000EFF00F0\r\n'], [b':FE05000F0000FE\r\n', b':FE05000FFF00FF\r\n'], [b':FE0F00000010020000E1\r\n', b':FE0F0000001002FFFFE3\r\n']]**

“”” Sainsmart 16-Channel 9-36v USB Relay Module (CH341 chip)(sku# 101-70-208) 1. Requires CH341 Windows driver installed (see http://wiki.sainsmart.com/index.php/101-70-208) 2. The hex table provided by Sainsmart requires converting to ASCII chars (see ‘usbrelay’ below) 3. Format of ‘usbrelay’ two-dimensional array is:

usbrelay = [row][ch-off, ch-on] so that the 2nd index selects ON/OFF value
while the 1st index selects the array row.

Example…
status = usbrelay[0][1] # row-0 (status) stat_ret = usbrelay[0][0] # row-0 (status return) ch_1_on = usbrelay[1][1] # row-1 (chan-1 off) ch_1_off = usbrelay[1][0] # row-1 (chan-1 off) ch_16_on = usbrelay[16][1] # row-16 (chan-16 on) ch_16_off = usbrelay[16][0] # row-16 (chan-16 off) all_on = usbrelay[17][1] # row-17 (all on) all_off = usbrelay[17][0] # row-17 (all off)

“””

Classes

MultiChannelRelay()	
SainSmartRelay(relaylabels, serial_port[, ...])	
SerialDevice(port[, baudrate, timeout, ...])	

AFL.automation.loading.SainSmartRelay.MultiChannelRelay

class AFL.automation.loading.SainSmartRelay.**MultiChannelRelay**

Bases: object

__init__()

Methods

__init__()	
getChannels(channels)	
setChannels(channels)	
toggleChannels(channels)	

setChannels(*channels*)

getChannels(*channels*)

toggleChannels(*channels*)

AFL.automation.loading.SainSmartRelay.SainSmartRelay

class AFL.automation.loading.SainSmartRelay.**SainSmartRelay**(*relaylabels*, *serial_port*, *timeout=0.5*)

Bases: *MultiChannelRelay*, *SerialDevice*

__init__(*relaylabels*, *serial_port*, *timeout=0.5*)

Init connection to a SainSmart 16-channel USB relay module.

Params: relaylabels (dict):

mapping of port id to load name, e.g. {0:'arm_up',1:'arm_down'}

serial_port (str):
port to connect to

Methods

__init__(relaylabels, serial_port[, timeout])	Init connection to a SainSmart 16-channel USB relay module.
getChannels([asid])	Read the current state of all channels
sendCommand(cmd[, response, questionmarkOK, ...])	
setAllChannelsOff()	
setChannels(channels)	Write a value (True, False) to the channels specified in channels
toggleChannels(channels)	

__init__(*relaylabels*, *serial_port*, *timeout=0.5*)

Init connection to a SainSmart 16-channel USB relay module.

Params: relaylabels (dict):

mapping of port id to load name, e.g. {0:'arm_up',1:'arm_down'}

serial_port (str):
port to connect to

setAllChannelsOff()

setChannels(*channels*)

Write a value (True, False) to the channels specified in channels

Parameters: channels (dict):

dict of channels, keys as either str (name) or int (id) to write to, vals as the value to write

getChannels(*asid=False*)

Read the current state of all channels

Parameters: asid (bool,default false): Dict keys should simply be the id, not the name.

Returns: (dict) key:value mappings of state.

toggleChannels(*channels*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.SainSmartRelay.SerialDevice

class AFL.automation.loading.SainSmartRelay.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

class AFL.automation.loading.SainSmartRelay.**SainSmartRelay**(*relaylabels*, *serial_port*, *timeout=0.5*)

__init__(*relaylabels*, *serial_port*, *timeout=0.5*)

Init connection to a SainSmart 16-channel USB relay module.

Params: relaylabels (dict):

mapping of port id to load name, e.g. {0:'arm_up',1:'arm_down'}

serial_port (str):
port to connect to

setAllChannelsOff()

setChannels(*channels*)

Write a value (True, False) to the channels specified in channels

Parameters: channels (dict):

dict of channels, keys as either str (name) or int (id) to write to, vals as the value to write

getChannels(*asid=False*)

Read the current state of all channels

Parameters: asid (bool,default false): Dict keys should simply be the id, not the name.

Returns: (dict) key:value mappings of state.

toggleChannels(*channels*)

AFL.automation.loading.SainSmartRelay.**usbrelay = [[b':FE0100200000FF\r\n', b':FE0100000010F1\r\n'], [b':FE0500000000FD\r\n', b':FE050000FF00FE\r\n'], [b':FE0500010000FC\r\n', b':FE050001FF00FD\r\n'], [b':FE0500020000FB\r\n', b':FE050002FF00FC\r\n'], [b':FE0500030000FA\r\n', b':FE050003FF00FB\r\n'], [b':FE0500040000F9\r\n', b':FE050004FF00FA\r\n'], [b':FE0500050000F8\r\n', b':FE050005FF00F9\r\n'], [b':FE0500060000F7\r\n', b':FE050006FF00F8\r\n'], [b':FE0500070000F6\r\n', b':FE050007FF00F7\r\n'], [b':FE0500080000F5\r\n', b':FE050008FF00F6\r\n'], [b':FE0500090000F4\r\n', b':FE050009FF00F5\r\n'], [b':FE05000A0000F3\r\n', b':FE05000AFF00F4\r\n'], [b':FE05000B0000F2\r\n', b':FE05000BFF00F3\r\n'], [b':FE05000C0000F1\r\n', b':FE05000CFF00F2\r\n'], [b':FE05000D0000F0\r\n', b':FE05000DFF00F1\r\n'], [b':FE05000E0000FF\r\n', b':FE05000EFF00F0\r\n'], [b':FE05000F0000FE\r\n', b':FE05000FFF00FF\r\n'], [b':FE0F000000100200000E1\r\n', b':FE0F0000001002FFFFE3\r\n']]**

“”” Sainsmart 16-Channel 9-36v USB Relay Module (CH341 chip)(sku# 101-70-208) 1. Requires CH341 Windows driver installed (see http://wiki.sainsmart.com/index.php/101-70-208) 2. The hex table provided by Sainsmart requires converting to ASCII chars (see ‘usbrelay’ below) 3. Format of ‘usbrelay’ two-dimensional array is:

usbrelay = [row][ch-off, ch-on] so that the 2nd index selects ON/OFF value
while the 1st index selects the array row.

Example…
status = usbrelay[0][1] # row-0 (status) stat_ret = usbrelay[0][0] # row-0 (status return) ch_1_on = usbrelay[1][1] # row-1 (chan-1 off) ch_1_off = usbrelay[1][0] # row-1 (chan-1 off) ch_16_on = usbrelay[16][1] # row-16 (chan-16 on) ch_16_off = usbrelay[16][0] # row-16 (chan-16 off) all_on = usbrelay[17][1] # row-17 (all on) all_off = usbrelay[17][0] # row-17 (all off)

“””

AFL.automation.loading.SampleCell

Classes

SampleCell()

AFL.automation.loading.SampleCell.SampleCell

class AFL.automation.loading.SampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()	
loadSample()	

loadSample()

class AFL.automation.loading.SampleCell.**SampleCell**

loadSample()

AFL.automation.loading.Sensor

Classes

DummySensor1([period, minval, maxval])	
DummySensor2([period, hi_value, lo_time, ...])	
Sensor([address, channel])	

AFL.automation.loading.Sensor.DummySensor1

class AFL.automation.loading.Sensor.**DummySensor1**(*period=2*, *minval=-5*, *maxval=5*)

Bases: Thread

__init__(*period=2*, *minval=-5*, *maxval=5*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__([period, minval, maxval])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
read()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*period=2*, *minval=-5*, *maxval=5*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

terminate()

run()

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

AFL.automation.loading.Sensor.DummySensor2

class AFL.automation.loading.Sensor.**DummySensor2**(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

Bases: Thread

__init__(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__([period, hi_value, lo_time, hi_time])	This constructor should always be called with keyword arguments.
driver_status()	
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
read()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

driver_status()

read()

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

AFL.automation.loading.Sensor.Sensor

class AFL.automation.loading.Sensor.**Sensor**(*address=1*, *channel=0*)

Bases: object

__init__(*address=1*, *channel=0*)

Methods

__init__([address, channel])
calibrate()
read()

__init__(*address=1*, *channel=0*)

calibrate()

read()

class AFL.automation.loading.Sensor.**Sensor**(*address=1*, *channel=0*)

__init__(*address=1*, *channel=0*)

calibrate()

read()

class AFL.automation.loading.Sensor.**DummySensor1**(*period=2*, *minval=-5*, *maxval=5*)

__init__(*period=2*, *minval=-5*, *maxval=5*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.loading.Sensor.**DummySensor2**(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

__init__(*period=0.1*, *hi_value=5*, *lo_time=15*, *hi_time=2*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

driver_status()

read()

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.loading.SensorCallbackThread

Classes

LoaderCommunication([load_client, load_object])	
SensorCallbackThread(poll[, period, daemon, ...])	
SimpleThresholdCB(poll, period[, window, ...])	
StopLoadCBv1(poll, period, load_client[, ...])	
StopLoadCBv2(poll, period[, load_client, ...])	

AFL.automation.loading.SensorCallbackThread.LoaderCommunication

class AFL.automation.loading.SensorCallbackThread.**LoaderCommunication**(*load_client=None*, *load_object=None*)

Bases: object

__init__(*load_client=None*, *load_object=None*)

Methods

__init__([load_client, load_object])	
getServerState()	
stopLoad()	

__init__(*load_client=None*, *load_object=None*)

getServerState()

stopLoad()

AFL.automation.loading.SensorCallbackThread.SensorCallbackThread

class AFL.automation.loading.SensorCallbackThread.**SensorCallbackThread**(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

Bases: Thread

__init__(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll[, period, daemon, filepath, ...])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal(signal)	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

update_status(*value*)

terminate()

process_signal(*signal*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

AFL.automation.loading.SensorCallbackThread.SimpleThresholdCB

class AFL.automation.loading.SensorCallbackThread.**SimpleThresholdCB**(*poll*, *period*, *window=5*, *threshold=1*)

Bases: *SensorCallbackThread*

__init__(*poll*, *period*, *window=5*, *threshold=1*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll, period[, window, threshold])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period*, *window=5*, *threshold=1*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

terminate()

update_status(*value*)

AFL.automation.loading.SensorCallbackThread.StopLoadCBv1

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv1**(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

Bases: *SensorCallbackThread*

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll, period, load_client[, ...])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

terminate()

update_status(*value*)

AFL.automation.loading.SensorCallbackThread.StopLoadCBv2

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv2**(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

Bases: *SensorCallbackThread*

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(poll, period[, load_client, ...])	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
process_signal()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	
update_status(value)	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

terminate()

update_status(*value*)

class AFL.automation.loading.SensorCallbackThread.**SensorCallbackThread**(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period=0.1*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

update_status(*value*)

terminate()

process_signal(*signal*)

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv1**(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period*, *load_client*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**StopLoadCBv2**(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

__init__(*poll*, *period*, *load_client=None*, *load_object=None*, *threshold_npts=20*, *threshold_v_step=1*, *threshold_std=2.5*, *timeout=120*, *min_load_time=3*, *loadstop_cooldown=2*, *post_detection_sleep=0.2*, *baseline_duration=2*, *trigger_on_end=False*, *instatrigger=True*, *daemon=True*, *filepath=None*, *data=None*, *sensorlabel=''*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**SimpleThresholdCB**(*poll*, *period*, *window=5*, *threshold=1*)

__init__(*poll*, *period*, *window=5*, *threshold=1*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

process_signal()

class AFL.automation.loading.SensorCallbackThread.**LoaderCommunication**(*load_client=None*, *load_object=None*)

__init__(*load_client=None*, *load_object=None*)

getServerState()

stopLoad()

AFL.automation.loading.SensorPollingThread

Classes

SensorPollingThread(sensor[, period, ...])	

AFL.automation.loading.SensorPollingThread.SensorPollingThread

class AFL.automation.loading.SensorPollingThread.**SensorPollingThread**(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

Bases: Thread

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(sensor[, period, callback, ...])	This constructor should always be called with keyword arguments.
alive()	
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
read()	
read_load_buffer()	
reset_load_buffer()	
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

read_load_buffer()

reset_load_buffer()

terminate()

alive()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class AFL.automation.loading.SensorPollingThread.SensorPollingThread(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

__init__(*sensor*, *period=0.1*, *callback=None*, *hv_pipe=None*, *window=None*, *filename=None*, *daemon=True*, *data=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read()

read_load_buffer()

reset_load_buffer()

terminate()

alive()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

AFL.automation.loading.SerialDevice

Classes

SerialDevice(port[, baudrate, timeout, ...])	

AFL.automation.loading.SerialDevice.SerialDevice

class AFL.automation.loading.SerialDevice.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

Exceptions

SerialCommsException	Raised when the system receives a serial response it can't parse, likely a garbled line

AFL.automation.loading.SerialDevice.SerialCommsException

exception AFL.automation.loading.SerialDevice.**SerialCommsException**

Raised when the system receives a serial response it can't parse, likely a garbled line

class AFL.automation.loading.SerialDevice.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.SyringePump

Classes

SyringePump()

AFL.automation.loading.SyringePump.SyringePump

class AFL.automation.loading.SyringePump.**SyringePump**

Bases: object

__init__()

Methods

__init__()
dispense(volume[, block])
emptySyringe()
getRate(rate)
setRate(rate)
stop()
withdraw(volume[, block])

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

class AFL.automation.loading.SyringePump.**SyringePump**

stop()

withdraw(*volume*, *block=True*)

dispense(*volume*, *block=True*)

setRate(*rate*)

getRate(*rate*)

emptySyringe()

AFL.automation.loading.Tubing

Classes

Tubing(specid, length)	

AFL.automation.loading.Tubing.Tubing

class AFL.automation.loading.Tubing.**Tubing**(*specid*, *length*)

Bases: object

__init__(*specid*, *length*)

length is in cm?

Methods

__init__(specid, length)	length is in cm?
volume()	returns volume in mL

Attributes

tubing

tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material': 'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in': 0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1517}]

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

class AFL.automation.loading.Tubing.**Tubing**(*specid*, *length*)

tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material': 'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in': 0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1517}]

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

AFL.automation.loading.TwoSelectorBlowoutSampleCell

Classes

Driver(name[, defaults, overrides])		
SampleCell()		
Tubing(specid, length)		
TwoSelectorBlowoutSampleCell(pump, ...)	selector,	Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).
defaultdict		defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.loading.TwoSelectorBlowoutSampleCell.Driver

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.loading.TwoSelectorBlowoutSampleCell.SampleCell

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**SampleCell**

Bases: object

__init__()

Methods

__init__()	
loadSample()	

loadSample()

AFL.automation.loading.TwoSelectorBlowoutSampleCell.Tubing

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**Tubing**(*specid*, *length*)

Bases: object

__init__(*specid*, *length*)

length is in cm?

Methods

__init__(specid, length)	length is in cm?
volume()	returns volume in mL

Attributes

tubing

tubing = [{'IDEXpart': 1530, 'ID_mm': 1.575, 'OD_in': 0.125, 'material': 'Tefzel', 'typeid': 1530}, {'IDEXpart': 1529, 'ID_mm': 0.254, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1529}, {'IDEXpart': 0, 'ID_mm': 2.92, 'OD_in': 0.1875, 'material': 'PVC', 'typeid': 1}, {'IDEXpart': 1517, 'ID_mm': 1, 'OD_in': 0.075, 'material': 'Tefzel', 'typeid': 1517}]

__init__(*specid*, *length*)

length is in cm?

volume()

returns volume in mL

AFL.automation.loading.TwoSelectorBlowoutSampleCell.TwoSelectorBlowoutSampleCell

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**TwoSelectorBlowoutSampleCell**(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level*, *overrides=None*)

Bases: *Driver*, *SampleCell*

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

Methods

__init__(pump, selector, blowselector[, ...])	ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells
blowOutCell([cellname, waittime])	
blowOutCellLegacy([cellname])	
catchToSyringe([sampleVolume])	
catchToWaste([sampleVolume])	
cellToWaste([cellname])	
deposit_obj(obj[, uid])	Store an object in the dropbox
drySyringe([blow, waittime])	transfer from air to waste, to push out any residual liquid.

continues on next page

Table 16 – continued from previous page

execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
loadSample([cellname, sampleVolume])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
reset_tank_levels([rinse, waste, cell_waste])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
rinseAll([cellname])	
rinseCatch()	
rinseCell([cellname])	
rinseCellFlood([cellname])	
rinseCellPull([cellname])	
rinseSyringe()	
setRinseLevel(vol)	
setWasteLevel(vol)	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
swish(vol)	

continues on next page

Table 16 – continued from previous page

transfer(source, dest, vol_source[, vol_dest])
unqueued()

Attributes

app
defaults

defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1, 'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2, 'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol': 1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0, 'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2, 'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3, 'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3, 'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol': 1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)

transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

setWasteLevel(*vol*)

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

AFL.automation.loading.TwoSelectorBlowoutSampleCell.defaultdict

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.loading.TwoSelectorBlowoutSampleCell.**TwoSelectorBlowoutSampleCell**(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

Class for a sample cell consisting of a pump and a one-to-many flow selector where the pump line holds sample (pulling and pushing as necessary) with a cell on a separate selector channel (in contrast to an inline selector cell where the cell is in the pump line).

@TODO: write support for multiple cells on separate channels (up to 6 cells on a 10-position selector)

```
defaults = {'blow_out_vol': 6, 'calibrated_catch_to_syringe_vol': 1.1,
'calibrated_syringe_to_cell_vol': 3.2, 'catch_empty_ffvol': 2,
'catch_to_selector_vol': 0.8796459430051422, 'cell_to_selector_vol':
1.9351768777756622, 'dry_vol_ml': 5, 'load_flow_delay': 10.0, 'load_speed': 10.0,
'ncells': 1, 'nrinses_catch': 2, 'nrinses_cell': 1, 'nrinses_cell_flood': 2,
'nrinses_syringe': 2, 'rinse_flow_delay': 3.0, 'rinse_prime_vol': 3,
'rinse_speed': 50.0, 'rinse_vol_catch_ml': 2, 'rinse_vol_ml': 3,
'selector_internal_vol': 0.0005067074790974977, 'syringe_to_selector_vol':
1.1625376729937205, 'thickness': None, 'to_waste_vol': 1}
```

__init__(*pump*, *selector*, *blowselector*, *rinse_tank_level=950*, *waste_tank_level=0*, *cell_waste_tank_level=0*, *overrides=None*)

ncells = number of connected cells (up to 6 cells with a 10-position flow selector, with four positions taken by load port, rinse, waste, and air) Name = the cell name, array with length = ncells

thickness = cell path length, to be incorporated into metadata, array with length = ncells

cell state if not 'clean', array with length = ncells

pump: a pump object supporting withdraw() and dispense() methods
e.g. pump = NE1KSyringePump(port,syringe_id_mm,syringe_volume)

selector: a selector object supporting string-based selectPort() method with options 'catch','cell','rinse','waste','air'
e.g. selector = ViciMultiposSelector(port,portlabels={'catch':1,'cell':2,'rinse':3,'waste':4,'air':5})

reset_tank_levels(*rinse=950*, *waste=0*, *cell_waste=0*)

property app

status()

transfer(*source*, *dest*, *vol_source*, *vol_dest=None*)

catchToSyringe(*sampleVolume=0*)

loadSample(*cellname='cell'*, *sampleVolume=0*)

catchToWaste(*sampleVolume=0.0*)

cellToWaste(*cellname='cell'*)

rinseSyringe()

drySyringe(*blow=True*, *waittime=1*)
transfer from air to waste, to push out any residual liquid.

if blow is True, additionally use a 1 s pulse of nitrogen to clear the syringe transfer line.

rinseCell(*cellname='cell'*)

rinseCellPull(*cellname='cell'*)

rinseCellFlood(*cellname='cell'*)

swish(*vol*)

blowOutCellLegacy(*cellname='cell'*)

blowOutCell(*cellname='cell'*, *waittime=20*)

rinseCatch()

rinseAll(*cellname='cell'*)

setRinseLevel(*vol*)

setWasteLevel(*vol*)

AFL.automation.loading.UltimusVPressureController

Classes

PressureController()	Abstract superclass for pressure controllers that provides timed dispensing
UltimusVPressureController(port[, baud])	

AFL.automation.loading.UltimusVPressureController.PressureController

class AFL.automation.loading.UltimusVPressureController.**PressureController**

Bases: object

Abstract superclass for pressure controllers that provides timed dispensing

__init__()

Methods

__init__()	
blockUntilStatusStopped([pollingdelay])	block execution until the controller finishes a dispense
dispenseRunning()	Returns true if a timed dispense is running, false otherwise.
ramp_dispense(dispense_start_pressure, ...)	Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*.
stop()	Abort the current timed dispense action.
timed_dispense(dispense_pressure, dispense_time)	Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*.

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

AFL.automation.loading.UltimusVPressureController.UltimusVPressureController

class AFL.automation.loading.UltimusVPressureController.**UltimusVPressureController**(*port*, *baud=115200*)

Bases: *PressureController*

__init__(*port*, *baud=115200*)

Initializes a DigitalOutPressureController

Params:

port (str): serial port to use

Methods

__init__(port[, baud])	Initializes a DigitalOutPressureController
blockUntilStatusStopped([pollingdelay])	block execution until the controller finishes a dispense
char_count(cmd)	
compute_checksum(cmd)	
dispenseRunning()	Returns true if a timed dispense is running, false otherwise.
package_cmd(cmd)	
ramp_dispense(dispense_start_pressure, ...)	Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*.
send_command(cmd)	
set_P(pressure)	pressure: pressure to set in psi
stop()	Abort the current timed dispense action.
timed_dispense(dispense_pressure, dispense_time)	Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*.

__init__(*port*, *baud=115200*)

Initializes a DigitalOutPressureController

Params:

port (str): serial port to use

compute_checksum(*cmd*)

char_count(*cmd*)

package_cmd(*cmd*)

send_command(*cmd*)

set_P(*pressure*)

pressure: pressure to set in psi

blockUntilStatusStopped(*pollingdelay=0.2*)

block execution until the controller finishes a dispense

dispenseRunning()

Returns true if a timed dispense is running, false otherwise.

ramp_dispense(*dispense_start_pressure*, *dispense_stop_pressure*, *dispense_time*, *const_time=0*, *block=True*)

Perform a pressure dispense with a linear ramp in pressure between *dispense_start_pressure* and *dispense_stop_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop(). If const_time is set, the last *const_time* seconds of the dispense will be at constant pressure, with the ramp occurring in the remaining time.

stop()

Abort the current timed dispense action.

timed_dispense(*dispense_pressure*, *dispense_time*, *block=True*)

Perform a pressure dispense at pressure *dispense_pressure*, stopping after *dispense_time*. This dispense can be interrupted by calling self.stop().

class AFL.automation.loading.UltimusVPressureController.**UltimusVPressureController**(*port*, *baud=115200*)

__init__(*port*, *baud=115200*)

Initializes a DigitalOutPressureController

Params:

port (str): serial port to use

compute_checksum(*cmd*)

char_count(*cmd*)

package_cmd(*cmd*)

send_command(*cmd*)

set_P(*pressure*)

pressure: pressure to set in psi

AFL.automation.loading.ViciMultiposSelector

Classes

FlowSelector()	
SerialDevice(port[, baudrate, timeout, ...])	
ViciMultiposSelector(port[, baudrate, ...])	

AFL.automation.loading.ViciMultiposSelector.FlowSelector

class AFL.automation.loading.ViciMultiposSelector.**FlowSelector**

Bases: object

__init__()

Methods

__init__()	
getPort()	
selectPort()	

getPort()

selectPort()

AFL.automation.loading.ViciMultiposSelector.SerialDevice

class AFL.automation.loading.ViciMultiposSelector.**SerialDevice**(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Bases: object

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

Methods

__init__(port[, baudrate, timeout, raw_writes])	
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=19200*, *timeout=0.5*, *raw_writes=False*)

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

AFL.automation.loading.ViciMultiposSelector.ViciMultiposSelector

class AFL.automation.loading.ViciMultiposSelector.**ViciMultiposSelector**(*port*, *baudrate=9600*, *portlabels=None*)

Bases: *SerialDevice*, *FlowSelector*

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

Methods

__init__(port[, baudrate, portlabels])	connect to valve and query the number of positions
getPort([as_str])	query the current selected position
selectPort(port[, direction, block])	moves the selector to portnum
sendCommand(cmd[, response, questionmarkOK, ...])	

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)

- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*, *block=True*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

Parameters

- **port** (*String or int*) – the name, or number, of the port to switch to
- **direction** (*String, default=None*) – direction "CW" or "CCW" to perform the move in
- **block** (*bool, default=True*) – block return until move completes

getPort(*as_str=False*)

query the current selected position

sendCommand(*cmd*, *response=True*, *questionmarkOK=False*, *timeout=-1*, *debug=False*)

class AFL.automation.loading.ViciMultiposSelector.**ViciMultiposSelector**(*port*, *baudrate=9600*, *portlabels=None*)

__init__(*port*, *baudrate=9600*, *portlabels=None*)

connect to valve and query the number of positions

Parameters

- **to** (*port - string describing the serial port the actuator is connected*)
- **use** (*baud - baudrate to*)
- **naming** (*portlabels - dict for smart port*) – 3,'instrument':4,'rinse':5,'waste':6}
- **{'sample'** (*of the form*) – 3,'instrument':4,'rinse':5,'waste':6}

selectPort(*port*, *direction=None*, *block=True*)

moves the selector to portnum

if direction is set to either "CW" or "CCW" it moves the actuator in that direction. if unset or other value, will move via most efficient route.

Parameters

- **port** (*String or int*) – the name, or number, of the port to switch to
- **direction** (*String, default=None*) – direction "CW" or "CCW" to perform the move in
- **block** (*bool, default=True*) – block return until move completes

getPort(*as_str=False*)

query the current selected position

6.2.4 Prepare

The Prepare module provides functionality for sample preparation and processing.

AFL.automation.prepare

AFL.automation.prepare

Functions

HD2OFactory(name[, phi_D2O, sld, properties])	Create a list of H2O/D2O solutions
compositionSweepFactory(name, components, ...)	
make_locs(slot, nrows, ncols)	
make_wellplate_locs(slot, size)	

AFL.automation.prepare.HD2OFactory

AFL.automation.prepare.**HD2OFactory**(*name*, *phi_D2O=None*, *sld=None*, *properties=None*)

Create a list of H2O/D2O solutions

AFL.automation.prepare.compositionSweepFactory

AFL.automation.prepare.**compositionSweepFactory**(*name*, *components*, *vary_components*, *lo*, *hi*, *num*, *logspace=False*, *properties=None*, *progress=None*)

AFL.automation.prepare.make_locs

AFL.automation.prepare.**make_locs**(*slot*, *nrows*, *ncols*)

AFL.automation.prepare.make_wellplate_locs

AFL.automation.prepare.**make_wellplate_locs**(*slot*, *size*)

Classes

ComponentDB([path])	
Deck()	
MassBalance()	
PipetteAction(source, dest, volume[, ...])	
Sample(name, target[, target_check, balancer])	
SampleSeries()	
Solute(name[, description, density, ...])	Specialization class for solute components
Solution(name, components[, properties])	
Solvent(name, density[, description, ...])	Specialization class for solute components

AFL.automation.prepare.ComponentDB

class AFL.automation.prepare.**ComponentDB**(*path='.afl/component.db.json'*)

Bases: object

__init__(*path='.afl/component.db.json'*)

Methods

__init__([path])	
add(name, preptype[, formula, density, sld, ...])	
add_interactive(name)	
read([path])	
remove(name[, write])	
write([path])	

__init__(*path='.afl/component.db.json'*)

read(*path=None*)

write(*path=None*)

add(*name*, *preptype*, *formula=None*, *density=None*, *sld=None*, *write=False*, *description=None*, *overwrite=False*)

add_interactive(*name*)

remove(*name*, *write=False*)

AFL.automation.prepare.Deck

class AFL.automation.prepare.**Deck**

Bases: object

__init__()

Methods

__init__()
add_catch(name, slot)
add_container(name, slot)
add_pipette(name, mount, tipracks)
add_stock(stock, location)
add_target(target[, location, name])
catch_sample(volume, source, dest[, ...])
get_components()
get_stock(name)
init_remote_connection(url[, home])
iterate_protocols()
make_align_script(filename[, load_last_sample])
make_mass_balance()
make_protocol([only_validated, flatten, ...])
make_sample_series([reset_sample_series])
make_script(filename[, load_last_sample])
reset_stocks()
reset_targets()
send_deck_config([debug_mode])
send_protocol([send_deck_config, debug_mode])
validate_sample_series([tolerance, ...])

__init__()

init_remote_connection(*url*, *home=False*)

catch_sample(*volume*, *source*, *dest*, *mix_before=None*, *debug_mode=False*)

send_deck_config(*debug_mode=False*)

send_protocol(*send_deck_config=True*, *debug_mode=False*)

add_pipette(*name*, *mount*, *tipracks*)

add_catch(*name*, *slot*)

add_container(*name*, *slot*)

add_stock(*stock*, *location*)

get_stock(*name*)

add_target(*target*, *location='target'*, *name=None*)

make_mass_balance()

reset_targets()

reset_stocks()

get_components()

make_sample_series(*reset_sample_series=False*)

validate_sample_series(*tolerance=0.0*, *print_report=True*, *progress=None*)

make_protocol(*only_validated=False*, *flatten=False*, *pipette_kw=None*)

iterate_protocols()

make_align_script(*filename*, *load_last_sample=True*)

make_script(*filename*, *load_last_sample=True*)

AFL.automation.prepare.MassBalance

class AFL.automation.prepare.**MassBalance**

Bases: object

__init__()

Methods

`__init__`()	
`add_stock`(stock, location)	
`balance_mass`()	
`calculate_bounds`([components, ...])	
`constrain_samples_conc`(constraints[, rtol])	
`copy`()	
`in_bounds`(points[, ternary])	
`make_grid_mask`([pts_per_row, ternary])	
`make_mass_fraction_matrix`()	
`make_target_component_masses`()	
`plot_bounds`([include_points])	
`plot_grid_mask`()	
`process_components`()	
`reset`()	
`reset_stocks`()	
`reset_targets`()	
`sample_composition_space`([pipette_min, ...])	Combine stock solutions to generate samples of possible target compositions
`set_target`(target, location)	

__init__()

copy()

add_stock(*stock*, *location*)

set_target(*target*, *location*)

reset_targets()

reset_stocks()

reset()

process_components()

make_mass_fraction_matrix()

make_target_component_masses()

balance_mass()

sample_composition_space(*pipette_min=<Quantity(5, 'microliter')>*, *stock_transfer_max=<Quantity(3500, 'microliter')>*, *grid_density=5*, *composition_data='frac'*)

Combine stock solutions to generate samples of possible target compositions

constrain_samples_conc(*constraints*, *rtol=0.05*)

calculate_bounds(*components=None*, *exclude_comps_below=None*, *fixed_comps=None*, *composition_data='samples_frac'*, *ternary=True*)

in_bounds(*points*, *ternary=True*)

make_grid_mask(*pts_per_row=100*, *ternary=True*)

plot_grid_mask()

plot_bounds(*include_points=True*)

AFL.automation.prepare.PipetteAction

class AFL.automation.prepare.**PipetteAction**(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

Bases: object

__init__(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

Methods

__init__(source, dest, volume[, source_loc, ...])	
emit_protocol()	
get_kwargs()	

__init__(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

emit_protocol()

get_kwargs()

AFL.automation.prepare.Sample

class AFL.automation.prepare.**Sample**(*name*, *target*, *target_check=None*, *balancer=None*)

Bases: object

__init__(*name*, *target*, *target_check=None*, *balancer=None*)

Methods

__init__(name, target[, target_check, balancer])
emit_protocol()

Attributes

balancer
target
target_check
target_loc

__init__(*name*, *target*, *target_check=None*, *balancer=None*)

emit_protocol()

property target

property target_check

property balancer

property target_loc

AFL.automation.prepare.SampleSeries

class AFL.automation.prepare.**SampleSeries**

Bases: object

__init__()

Methods

`__init__`()	
`add_sample`(sample)	
`mass_totals_component`([only_validated])	
`mass_totals_stock`([only_validated])	
`reset`()	
`shuffle`()	

__init__()

reset()

shuffle()

add_sample(*sample*)

mass_totals_stock(*only_validated=True*)

mass_totals_component(*only_validated=True*)

AFL.automation.prepare.Solute

class AFL.automation.prepare.**Solute**(*name*, *description=None*, *density=None*, *formula=None*, *sld=None*)

Bases: *Component*

Specialization class for solute components

__init__(*name*, *description=None*, *density=None*, *formula=None*, *sld=None*)

Methods

`__init__`(name[, description, density, ...])	
`copy`()	
`emit`()	
`set_mass`(value)	Setter for inline mass changes
`set_volume`(volume)	Setter for inline volume changes

Attributes

density
formula
is_solute
is_solvent
mass
moles
sld
volume

__init__(*name*, *description=None*, *density=None*, *formula=None*, *sld=None*)

property density

property formula

property sld

property volume

__hash__()

Needed so Components can be dictionary keys

__iter__()

Dummy iterator to mimic behavior of Mixture.

copy()

emit()

property is_solute

property is_solvent

property mass

property moles

set_mass(*value*)

Setter for inline mass changes

set_volume(*volume*)

Setter for inline volume changes

AFL.automation.prepare.Solution

class AFL.automation.prepare.**Solution**(*name*, *components*, *properties=None*)

Bases: object

__init__(*name*, *components*, *properties=None*)

Methods

__init__(name, components[, properties])	
add_component_from_name(name[, properties, ...])	
contains(name)	
copy([name])	
from_dict(in_dict)	
measure_out(amount[, deplete])	Create solution with identical composition at new total mass/volume
rename_component(old_name, new_name[, inplace])	
set_mass(value)	Setter for inline mass changes
set_properties_from_dict([properties, inplace])	
set_volume(value)	Setter for inline volume changes
to_dict()	

Attributes

concentration	
mass	Total mass of mixture.
mass_fraction	Mass fraction of components in mixture
molarity	
size	
solutes	
solvent_density	
solvent_mass	
solvent_sld	
solvent_volume	
solvents	
volume	Total volume of mixture.
volume_fraction	Volume fraction of solvents in mixture

__init__(*name*, *components*, *properties=None*)

__hash__()

Needed so Solutions can be dictionary keys

to_dict()

classmethod from_dict(*in_dict*)

add_component_from_name(*name*, *properties=None*, *inplace=False*)

set_properties_from_dict(*properties=None*, *inplace=False*)

rename_component(*old_name*, *new_name*, *inplace=False*)

copy(*name=None*)

contains(*name*)

property size

property solutes

property solvents

__eq__(*other*)

'Compare the mass,volume, and composition of two mixtures

property mass

Total mass of mixture.

set_mass(*value*)

Setter for inline mass changes

property volume

Total volume of mixture. Only solvents are included in volume calculation

set_volume(*value*)

Setter for inline volume changes

property solvent_sld

property solvent_density

property solvent_volume

property solvent_mass

property mass_fraction

Mass fraction of components in mixture

Returns

- **mass_fraction** (*dict*)
- *Component mass fractions*

property volume_fraction

Volume fraction of solvents in mixture

Returns

- **solvent_fraction** (*dict*)
- *Component mass fractions*

property concentration

property molarity

measure_out(*amount*, *deplete=False*)

Create solution with identical composition at new total mass/volume

AFL.automation.prepare.Solvent

class AFL.automation.prepare.**Solvent**(*name*, *density*, *description=None*, *formula=None*, *sld=None*)

Bases: *Component*

Specialization class for solute components

__init__(*name*, *density*, *description=None*, *formula=None*, *sld=None*)

Methods

`__init__`(name, density[, description, ...])	
`copy`()	
`emit`()	
`set_mass`(value)	Setter for inline mass changes
`set_volume`(value)	Setter for inline volume changes

Attributes

`density`
`formula`
`is_solute`
`is_solvent`
`mass`
`moles`
`sld`
`volume`

__init__(*name*, *density*, *description=None*, *formula=None*, *sld=None*)

property density

property formula

property sld

__hash__()

Needed so Components can be dictionary keys

__iter__()

Dummy iterator to mimic behavior of Mixture.

copy()

emit()

property is_solute

property is_solvent

property mass

property moles

set_mass(*value*)
Setter for inline mass changes

set_volume(*value*)
Setter for inline volume changes

property volume

Modules

Component
DeckBuilderWidget
Dummy_OT2_Driver
OT2Client
PrepType
PrepareWidget
SampleSeriesWidget
StockBuilderWidget
SweepBuilderWidget
factory
utilities

AFL.automation.prepare.Component

Functions

enforce_units(value, unit_type)	Ensure that a number has units and convert to the default_units

AFL.automation.prepare.Component.enforce_units

AFL.automation.prepare.Component.**enforce_units**(*value*, *unit_type*)
Ensure that a number has units and convert to the default_units

Classes

Component(name, description)	Base class for all materials
PrepType(value[, names, module, qualname, ...])	

AFL.automation.prepare.Component.Component

class AFL.automation.prepare.Component.**Component**(*name*, *description*)

Bases: object

Base class for all materials

This class defines all of the basic properties and methods to be shared across material objects

__init__(*name*, *description*)

Methods

__init__(name, description)	
copy()	
emit()	
set_mass(value)	Setter for inline mass changes
set_volume(value)	Setter for inline volume changes

Attributes

density
formula
is_solute
is_solvent
mass
moles
sld
volume

__init__(*name*, *description*)

emit()

__hash__()

Needed so Components can be dictionary keys

copy()

__iter__()

Dummy iterator to mimic behavior of Mixture.

property mass

set_mass(*value*)

Setter for inline mass changes

property volume

set_volume(*value*)

Setter for inline volume changes

property density

property formula

property moles

property sld

property is_solute

property is_solvent

AFL.automation.prepare.Component.PrepType

class AFL.automation.prepare.Component.**PrepType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *Enum*

__init__(**args*, ***kwds*)

Attributes

BaseComponent
BaseMixture
Solute
Solvent
Solution

BaseComponent = 1

BaseMixture = 2

Solute = 3

Solvent = 4

Solution = 5

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

Exceptions

ParseException(pstr[, loc, msg, elem])	Exception thrown when a parse expression doesn't match the input string

AFL.automation.prepare.Component.ParseException

exception AFL.automation.prepare.Component.**ParseException**(*pstr: str*, *loc: int = 0*, *msg: str | None = None*, *elem=None*)

Exception thrown when a parse expression doesn't match the input string

Example:

```
integer = Word(nums).set_name("integer")
try:
    integer.parse_string("ABC")
except ParseException as pe:
    print(pe, f"column: {pe.column}")
```

prints:

```
Expected integer, found 'ABC'  (at char 0), (line:1, col:1) column: 1
```

loc: int

msg: str

pstr: str

parser_element: Any

args: tuple[str, int, str | None]

class AFL.automation.prepare.Component.**Component**(*name*, *description*)

Base class for all materials

This class defines all of the basic properties and methods to be shared across material objects

__init__(*name*, *description*)

emit()

__hash__()
Needed so Components can be dictionary keys

copy()

__iter__()
Dummy iterator to mimic behavior of Mixture.

property mass

set_mass(*value*)
Setter for inline mass changes

property volume

set_volume(*value*)
Setter for inline volume changes

property density

property formula

property moles

property sld

property is_solute

property is_solvent

AFL.automation.prepare.DeckBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.prepare.DeckBuilderWidget.sqrt

AFL.automation.prepare.DeckBuilderWidget.**sqrt**(*x*, /)
Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
Client([ip, port, username, interactive])	Communicate with APIServer
DeckBuilderWidget()	
DeckBuilderWidget_Model()	
DeckBuilderWidget_View()	
Dropdown(**kwargs)	Allows you to select a single item from a dropdown.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

AFL.automation.prepare.DeckBuilderWidget.Button

class AFL.automation.prepare.DeckBuilderWidget.**Button**(***kwargs: Any*)

Bases: DOMWidget, CoreWidget

Button widget.

This widget has an *on_click* method that allows you to listen for the user clicking on the button. The click event itself is stateless.

Parameters

- **description** (*str*) – description displayed on the button
- **icon** (*str*) – font-awesome icon names, without the 'fa-' prefix
- **disabled** (*bool*) – whether user interaction is enabled

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
click()	Programmatically trigger a click event.

continues on next page

Table 17 – continued from previous page

close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_click(callback[, remove])	Register a callback to execute when the button is clicked.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

button_style	Use a predefined styling for the button.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Button label.
disabled	Enable or disable user changes.
icon	Font-awesome icon names, without the 'fa-' prefix.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

description

Button label.

disabled

Enable or disable user changes.

icon

Font-awesome icon names, without the 'fa-' prefix.

button_style

Use a predefined styling for the button.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(***kwargs*)

Public constructor

on_click(*callback*, *remove=False*)

Register a callback to execute when the button is clicked.

The callback will be called with one argument, the clicked button widget instance.

Parameters
remove (*bool (optional)*) – Set to true to remove the callback from the list of callbacks.

click()

Programmatically trigger a click event.

This will call the callbacks registered to the clicked button widget instance.

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(**traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.Checkbox

class AFL.automation.prepare.DeckBuilderWidget.**Checkbox**(***kwargs: Any*)

Bases: _Bool

Displays a boolean *value* in the form of a checkbox.

Parameters

- **value** (*{True,False}*) – value of the checkbox: True-checked, False-unchecked
- **description** (*str*) – description displayed next to the checkbox
- **indent** (*{True,False}*) – indent the control to align with other controls with a description. The style.description_width attribute controls this width for consistence with other controls.

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.

continues on next page

Table 18 – continued from previous page

get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes.
indent	Indent the control to align with other controls with a description.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Bool value
widget_types	
widgets	

indent

Indent the control to align with other controls with a description.

style

Styling customizations

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Bool value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.Client

class AFL.automation.prepare.DeckBuilderWidget.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	

continues on next page

Table 19 – continued from previous page

get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.prepare.DeckBuilderWidget.DeckBuilderWidget

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget**

Bases: object

__init__()

Methods

__init__()
build_deck_object()
get_deck_config()
load_cb(event)
save_cb(event)
send_deck_cb(event)
set_deck_config(config)
start()
update_deck_graphic_cb(event)

__init__()

build_deck_object()

get_deck_config()

set_deck_config(*config*)

send_deck_cb(*event*)

save_cb(*event*)

load_cb(*event*)

update_deck_graphic_cb(*event*)

start()

AFL.automation.prepare.DeckBuilderWidget.DeckBuilderWidget_Model

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_Model**

Bases: object

__init__()

Methods

__init__()	
build_deck_object(config)	
send_deck_config([pi_ip, align_script])	

__init__()

send_deck_config(*pi_ip='piot2'*, *align_script='/home/nistoroboto/align.py'*)

build_deck_object(*config*)

AFL.automation.prepare.DeckBuilderWidget.DeckBuilderWidget_View

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_View**

Bases: object

__init__()

Methods

__init__()	
create_expanded_button(description, button_style)	
draw_deck()	
start()	

__init__()

create_expanded_button(*description*, *button_style*)

draw_deck()

start()

AFL.automation.prepare.DeckBuilderWidget.Dropdown

class AFL.automation.prepare.DeckBuilderWidget.**Dropdown**(***kwargs: Any*)

Bases: _Selection

Allows you to select a single item from a dropdown.

Parameters

- **options** (*list*) – The options for the dropdown. This can either be a list of values, e.g. `['Galileo', 'Brahe', 'Hubble']` or `[0, 1, 2]`, a list of (label, value) pairs, e.g. `[('Galileo', 0), ('Brahe', 1), ('Hubble', 2)]`, or a Mapping between labels and values, e.g., `{'Galileo': 0, 'Brahe': 1, 'Hubble': 2}`.

- **index** (*int*) – The index of the current selection.
- **value** (*any*) – The value of the current selection. When programmatically setting the value, a reverse lookup is performed among the options to check that the value is valid. The reverse lookup uses the equality operator by default, but another predicate may be provided via the `equals` keyword argument. For example, when dealing with numpy arrays, one may set `equals=np.array_equal`.
- **label** (*str*) – The label corresponding to the selected value.
- **disabled** (*bool*) – Whether to disable user changes.
- **description** (*str*) – Label for this input group. This should be a string describing the widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.

continues on next page

Table 20 – continued from previous page

on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
index	Selected index
keys	The traits which are synced.
label	Selected label
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
options	Iterable of values, (label, value) pairs, or Mapping between labels and values that the user can select.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Selected value
widget_types	
widgets	

__del__()

Object disposal

__init__(**args*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

index

Selected index

keys

The traits which are synced.

label

Selected label

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

options

Iterable of values, (label, value) pairs, or Mapping between labels and values that the user can select.

The labels are the strings that will be displayed in the UI, representing the actual Python choices, and should be unique.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

style

Styling customizations

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Selected value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.HBox

class AFL.automation.prepare.DeckBuilderWidget.**HBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets horizontally using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Horizontal Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.HBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.

continues on next page

Table 21 – continued from previous page

trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: *Any*

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys:

* owner : the HasTraits instance * old : the old value of the modified trait attribute * new : the new value of the modified trait attribute * name : the name of the modified trait attribute.

- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.Label

class AFL.automation.prepare.DeckBuilderWidget.**Label**(***kwargs: Any*)

Bases: _String

Label widget.

It also renders math inside the string *value* as Latex (requires $ $ or $$ $$ and similar latex tags).

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.

continues on next page

Table 22 – continued from previous page

remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

`style`

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(**kwargs: *Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(*names: *str*, **metadata: *Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: *str* | *None* = *None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: *str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: *str*, *key: *str*, *default: *Any* = *None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: *Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.Layout

class AFL.automation.prepare.DeckBuilderWidget.**Layout**(***kwargs: Any*)

Bases: `Widget`

Layout specification

Defines a layout that can be expressed using CSS. Supports a subset of https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

When a property is also accessible via a shorthand property, we only expose the shorthand.

For example: - `flex-grow`, `flex-shrink` and `flex-basis` are bound to `flex`. - `flex-wrap` and `flex-direction` are bound to `flex-flow`. - `margin-[top/bottom/left/right]` values are bound to `margin`, etc.

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_traits(**traits)	Dynamically add trait attributes to the Widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.

continues on next page

Table 23 – continued from previous page

trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

align_content	The align-content CSS attribute.
align_items	The align-items CSS attribute.
align_self	The align-self CSS attribute.
border	*border* property getter.
border_bottom	The border bottom CSS attribute.
border_left	The border left CSS attribute.
border_right	The border right CSS attribute.
border_top	The border top CSS attribute.
bottom	The bottom CSS attribute.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
display	The display CSS attribute.
flex	The flex CSS attribute.
flex_flow	The flex-flow CSS attribute.
grid_area	The grid-area CSS attribute.
grid_auto_columns	The grid-auto-columns CSS attribute.
grid_auto_flow	The grid-auto-flow CSS attribute.
grid_auto_rows	The grid-auto-rows CSS attribute.
grid_column	The grid-column CSS attribute.
grid_gap	The grid-gap CSS attribute.
grid_row	The grid-row CSS attribute.
grid_template_areas	The grid-template-areas CSS attribute.
grid_template_columns	The grid-template-columns CSS attribute.
grid_template_rows	The grid-template-rows CSS attribute.
height	The height CSS attribute.
justify_content	The justify-content CSS attribute.
justify_items	The justify-items CSS attribute.
keys	The traits which are synced.
left	The left CSS attribute.
log	A trait whose value must be an instance of a specified class.
margin	The margin CSS attribute.
max_height	The max-height CSS attribute.
max_width	The max-width CSS attribute.
min_height	The min-height CSS attribute.
min_width	The min-width CSS attribute.
model_id	Gets the model id of this widget.
object_fit	The object-fit CSS attribute.
object_position	The object-position CSS attribute.
order	The order CSS attribute.
overflow	The overflow CSS attribute.
padding	The padding CSS attribute.

continues on next page

Table 24 – continued from previous page

right	The right CSS attribute.
top	The top CSS attribute.
visibility	The visibility CSS attribute.
widget_types	
widgets	
width	The width CSS attribute.

align_content
The align-content CSS attribute.

align_items
The align-items CSS attribute.

align_self
The align-self CSS attribute.

border_top
The border top CSS attribute.

border_right
The border right CSS attribute.

border_bottom
The border bottom CSS attribute.

border_left
The border left CSS attribute.

bottom
The bottom CSS attribute.

display
The display CSS attribute.

flex
The flex CSS attribute.

flex_flow
The flex-flow CSS attribute.

height
The height CSS attribute.

justify_content
The justify-content CSS attribute.

justify_items
The justify-items CSS attribute.

left
The left CSS attribute.

margin
The margin CSS attribute.

`max_height`
: The max-height CSS attribute.

`max_width`
: The max-width CSS attribute.

`min_height`
: The min-height CSS attribute.

`min_width`
: The min-width CSS attribute.

`overflow`
: The overflow CSS attribute.

`order`
: The order CSS attribute.

`padding`
: The padding CSS attribute.

`right`
: The right CSS attribute.

`top`
: The top CSS attribute.

`visibility`
: The visibility CSS attribute.

`width`
: The width CSS attribute.

`object_fit`
: The object-fit CSS attribute.

`object_position`
: The object-position CSS attribute.

`grid_auto_columns`
: The grid-auto-columns CSS attribute.

`grid_auto_flow`
: The grid-auto-flow CSS attribute.

`grid_auto_rows`
: The grid-auto-rows CSS attribute.

`grid_gap`
: The grid-gap CSS attribute.

`grid_template_rows`
: The grid-template-rows CSS attribute.

`grid_template_columns`
: The grid-template-columns CSS attribute.

grid_template_areas

The grid-template-areas CSS attribute.

grid_row

The grid-row CSS attribute.

grid_column

The grid-column CSS attribute.

grid_area

The grid-area CSS attribute.

__init__(***kwargs*)

Public constructor

property border

border property getter. Return the common value of all side borders if they are identical. Otherwise return None.

__del__()

Object disposal

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.Text

class AFL.automation.prepare.DeckBuilderWidget.**Text**(***kwargs: Any*)

Bases: _String

Single line textbox widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.

continues on next page

Table 25 – continued from previous page

hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_submit(callback[, remove])	(Un)Register a callback to handle text submission.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(**args*, ***kwargs*)

Public constructor

on_submit(*callback*, *remove=False*)

(Un)Register a callback to handle text submission.

Triggered when the user clicks enter.

Parameters

- **callback** (*callable*) – Will be called with exactly one argument: the Widget instance
- **remove** (*bool (optional)*) – Whether to unregister the callback

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.DeckBuilderWidget.VBox

class AFL.automation.prepare.DeckBuilderWidget.**VBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets vertically using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Vertical Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.VBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the frontend.

continues on next page

Table 26 – continued from previous page

send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.

- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget**

__init__()

build_deck_object()

get_deck_config()

set_deck_config(*config*)

send_deck_cb(*event*)

save_cb(*event*)

load_cb(*event*)

update_deck_graphic_cb(*event*)

start()

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_Model**

__init__()

send_deck_config(*pi_ip='piot2'*, *align_script='/home/nistoroboto/align.py'*)

build_deck_object(*config*)

class AFL.automation.prepare.DeckBuilderWidget.**DeckBuilderWidget_View**

__init__()

create_expanded_button(*description*, *button_style*)

draw_deck()

start()

AFL.automation.prepare.Dummy_OT2_Driver

Classes

Driver(name[, defaults, overrides])	
Dummy_OT2_Driver([overrides])	

AFL.automation.prepare.Dummy_OT2_Driver.Driver

class AFL.automation.prepare.Dummy_OT2_Driver.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.prepare.Dummy_OT2_Driver.Dummy_OT2_Driver

class AFL.automation.prepare.Dummy_OT2_Driver.**Dummy_OT2_Driver**(*overrides=None*)

Bases: *Driver*

__init__(*overrides=None*)

Methods

__init__([overrides])	
add_prep_targets(targets[, reset])	
deactivate_temp(slot)	Disablea tempdeck in slot slot
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_labware(slot)	
get_object(name[, serialize])	
get_prep_target()	
get_sample()	
get_shake_latch_status()	
get_shake_rpm()	
get_shaker_temp()	
get_temp(slot)	Get the temperature of a tempdeck in slot slot
get_wells(locs)	
home(**kwargs)	
latch_shaker()	
load_instrument(name, mount, tip_rack_slots, ...)	
load_labware(name, slot[, module])	
parse_well(loc)	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute

continues on next page

Table 27 – continued from previous page

queued()	
quickbar()	
reset_prep_targets()	
reset_sample()	
reset_tipracks([mount])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_aspirate_rate([rate])	
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_dispense_rate([rate])	
set_gantry_speed([speed])	
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
set_shake(rpm)	
set_shaker_temp(temp)	
set_temp(slot, temp)	Set the temperature of a tempdeck in slot slot
status()	
stop_shake()	
transfer(source, dest, volume, *args, **kwargs)	
unlatch_shaker()	
unqueued()	

Attributes

defaults

defaults = {'execute_delay': 1}

__init__(*overrides=None*)

reset_prep_targets()

add_prep_targets(*targets*, *reset=False*)

get_prep_target()

status()

reset_tipracks(*mount='both'*)

home(***kwargs*)

parse_well(*loc*)

get_wells(*locs*)

get_labware(*slot*)

load_labware(*name*, *slot*, *module=None*, ***kwargs*)

set_temp(*slot*, *temp*)
Set the temperature of a tempdeck in slot slot

get_temp(*slot*)
Get the temperature of a tempdeck in slot slot

deactivate_temp(*slot*)
Disablea tempdeck in slot slot

set_shake(*rpm*)

stop_shake()

set_shaker_temp(*temp*)

unlatch_shaker()

latch_shaker()

get_shaker_temp()

get_shake_rpm()

get_shake_latch_status()

load_instrument(*name*, *mount*, *tip_rack_slots*, ***kwargs*)

transfer(*source*, *dest*, *volume*, **args*, ***kwargs*)

set_aspirate_rate(*rate=150*)

set_dispense_rate(*rate=300*)

set_gantry_speed(*speed=400*)

deposit_obj(*obj*, *uid=None*)
Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.prepare.Dummy_OT2_Driver.**Dummy_OT2_Driver**(*overrides=None*)

defaults = {'execute_delay': 1}

__init__(*overrides=None*)

reset_prep_targets()

add_prep_targets(*targets*, *reset=False*)

get_prep_target()

status()

reset_tipracks(*mount='both'*)

home(***kwargs*)

parse_well(*loc*)

get_wells(*locs*)

get_labware(*slot*)

load_labware(*name*, *slot*, *module=None*, ***kwargs*)

set_temp(*slot*, *temp*)

Set the temperature of a tempdeck in slot slot

get_temp(*slot*)

Get the temperature of a tempdeck in slot slot

deactivate_temp(*slot*)

Disablea tempdeck in slot slot

set_shake(*rpm*)

stop_shake()

set_shaker_temp(*temp*)

unlatch_shaker()

latch_shaker()

get_shaker_temp()

get_shake_rpm()

get_shake_latch_status()

load_instrument(*name*, *mount*, *tip_rack_slots*, ***kwargs*)

transfer(*source*, *dest*, *volume*, **args*, ***kwargs*)

set_aspirate_rate(*rate=150*)

set_dispense_rate(*rate=300*)

set_gantry_speed(*speed=400*)

AFL.automation.prepare.OT2Client

Classes

Client([ip, port, username, interactive])	Communicate with APIServer
OT2Client([ip, port, username, interactive])	Communicate with AFL-automation server on OT-2
PipetteAction(source, dest, volume[, ...])	

AFL.automation.prepare.OT2Client.Client

class AFL.automation.prepare.OT2Client.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	

continues on next page

Table 28 – continued from previous page

get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.prepare.OT2Client.OT2Client

class AFL.automation.prepare.OT2Client.**OT2Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: *Client*

Communicate with AFL-automation server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the AFL-automation server

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
aspirate_rate(rate)	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
dispense_rate(rate)	
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	

continues on next page

Table 29 – continued from previous page

halt()	
home()	
load_instrument(name, mount, tip_rack_slots)	
load_labware(name, slot)	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
reset_tipracks([mount])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
transfer(source, dest, volume[, interactive])	Transfer fluid from one location to another
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

transfer(*source*, *dest*, *volume*, *interactive=None*, ***kwargs*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str Source wells to transfer from. Wells should be specified as three*) – character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.

- **volume** (*float*) – volume of fluid to transfer in microliters

reset_tipracks(*mount='both'*)

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

aspirate_rate(*rate*)

dispense_rate(*rate*)

home()

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

clear_history()

clear_queue()

debug(*state*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

driver_status()

enqueue(*interactive=None*, ***kwargs*)

enqueued_base(***kwargs*)

classmethod from_server_name(*server_name*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_driver_object(*name*)

get_object(*name*, *serialize=True*)

get_queue()

get_queued_commmands(*inherit_commands=True*)

get_quickbar()

get_server_time()

get_unqueued_commmands(*inherit_commands=True*)

halt()

logged_in()

login(*username*, *populate_commands=True*)

move_item(*uuid*, *pos*)

pause(*state*)

query_driver(***kwargs*)

queue_state()

remove_item(*uuid*)

reset_queue_daemon()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

server_cmd(*cmd*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

set_driver_object(***kw*)

set_object(*serialize=True*, ***kw*)

unqueued_base(***kwargs*)

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

AFL.automation.prepare.OT2Client.PipetteAction

class AFL.automation.prepare.OT2Client.**PipetteAction**(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

Bases: object

__init__(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

Methods

__init__(source, dest, volume[, source_loc, ...])
emit_protocol()
get_kwargs()

__init__(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*, *aspirate_rate=None*, *dispense_rate=None*, *mix_aspirate_rate=None*, *mix_dispense_rate=None*, *mix_before=None*, *mix_after=None*, *blow_out=False*, *post_aspirate_delay=0.0*, *post_dispense_delay=0.0*, *aspirate_equilibration_delay=0.0*, *drop_tip=True*, *force_new_tip=False*, *to_top=True*, *to_center=False*, *to_top_z_offset=0*, *fast_mixing=False*)

emit_protocol()

get_kwargs()

class AFL.automation.prepare.OT2Client.**OT2Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Communicate with AFL-automation server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the AFL-automation server

transfer(*source*, *dest*, *volume*, *interactive=None*, ***kwargs*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str Source wells to transfer from. Wells should be specified as three*) – character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

reset_tipracks(*mount='both'*)

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

aspirate_rate(*rate*)

dispense_rate(*rate*)

home()

AFL.automation.prepare.PrepType

Functions

makeRegistar()
prepRegistrar(prepType)

AFL.automation.prepare.PrepType.makeRegistar

AFL.automation.prepare.PrepType.**makeRegistar**()

AFL.automation.prepare.PrepType.prepRegistrar

AFL.automation.prepare.PrepType.**prepRegistrar**(*prepType*)

Classes

Enum(value[, names, module, qualname, type, ...])	Create a collection of name/value pairs.
PrepType(value[, names, module, qualname, ...])	
auto([value])	Instances are replaced with an appropriate value in Enum class suites.

AFL.automation.prepare.PrepType.Enum

class AFL.automation.prepare.PrepType.**Enum**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: object

Create a collection of name/value pairs.

Example enumeration:

```
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
...     GREEN = 3
```

Access them by:

- attribute access:

```
>>> Color.RED
<Color.RED: 1>
```

- value lookup:

```
>>> Color(1)
<Color.RED: 1>
```

- name lookup:

```
>>> Color['RED']
<Color.RED: 1>
```

Enumerations can be iterated over, and know how many members they have:

```
>>> len(Color)
3
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Methods can be added to enumerations, and members can have their own attributes – see the documentation for details.

__init__(**args*, ***kwds*)

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

AFL.automation.prepare.PrepType.PrepType

class AFL.automation.prepare.PrepType.**PrepType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *Enum*

__init__(**args*, ***kwds*)

Attributes

BaseComponent
BaseMixture
Solute
Solvent
Solution

BaseComponent = 1

BaseMixture = 2

Solute = 3

Solvent = 4

Solution = 5

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

AFL.automation.prepare.PrepType.auto

class AFL.automation.prepare.PrepType.**auto**(*value=_auto_null*)

Bases: object

Instances are replaced with an appropriate value in Enum class suites.

__init__(*value=_auto_null*)

Methods

__init__([value])	

__init__(*value=_auto_null*)

class AFL.automation.prepare.PrepType.**PrepType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

BaseComponent = 1

BaseMixture = 2

Solute = 3

Solvent = 4

Solution = 5

AFL.automation.prepare.PrepareWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.prepare.PrepareWidget.sqrt

AFL.automation.prepare.PrepareWidget.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
DeckBuilderWidget()	
Dropdown(**kwargs)	Allows you to select a single item from a dropdown.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
PrepareWidget()	
PrepareWidget_Model()	
PrepareWidget_View()	
SampleSeriesWidget(deck)	
StockBuilderWidget(deck)	
SweepBuilderWidget(deck)	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

AFL.automation.prepare.PrepareWidget.Button

class AFL.automation.prepare.PrepareWidget.**Button**(***kwargs: Any*)

Bases: DOMWidget, CoreWidget

Button widget.

This widget has an *on_click* method that allows you to listen for the user clicking on the button. The click event itself is stateless.

Parameters

- **description** (*str*) – description displayed on the button
- **icon** (*str*) – font-awesome icon names, without the 'fa-' prefix
- **disabled** (*bool*) – whether user interaction is enabled

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.

continues on next page

Table 30 – continued from previous page

class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
click()	Programmatically trigger a click event.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_click(callback[, remove])	Register a callback to execute when the button is clicked.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.

continues on next page

Table 30 – continued from previous page

traits(**metadata)	Get a dict of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

button_style	Use a predefined styling for the button.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Button label.
disabled	Enable or disable user changes.
icon	Font-awesome icon names, without the 'fa-' prefix.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

description

Button label.

disabled

Enable or disable user changes.

icon

Font-awesome icon names, without the 'fa-' prefix.

button_style

Use a predefined styling for the button.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(***kwargs*)

Public constructor

on_click(*callback*, *remove=False*)

Register a callback to execute when the button is clicked.

The callback will be called with one argument, the clicked button widget instance.

Parameters
remove (*bool (optional)*) – Set to true to remove the callback from the list of callbacks.

click()

Programmatically trigger a click event.

This will call the callbacks registered to the clicked button widget instance.

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False,

then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.Checkbox

class AFL.automation.prepare.PrepareWidget.**Checkbox**(***kwargs: Any*)

Bases: _Bool

Displays a boolean *value* in the form of a checkbox.

Parameters

- **value** (*{True,False}*) – value of the checkbox: True-checked, False-unchecked
- **description** (*str*) – description displayed next to the checkbox
- **indent** (*{True,False}*) – indent the control to align with other controls with a description. The style.description_width attribute controls this width for consistence with other controls.

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.

continues on next page

Table 31 – continued from previous page

class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a dict of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A dict of trait names and their values.
traits(**metadata)	Get a dict of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes.
indent	Indent the control to align with other controls with a description.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Bool value
widget_types	
widgets	

indent

Indent the control to align with other controls with a description.

style

Styling customizations

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(****metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(****metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(****metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Bool value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.DeckBuilderWidget

class AFL.automation.prepare.PrepareWidget.**DeckBuilderWidget**

Bases: object

__init__()

Methods

__init__()
build_deck_object()
get_deck_config()
load_cb(event)
save_cb(event)
send_deck_cb(event)
set_deck_config(config)
start()
update_deck_graphic_cb(event)

__init__()

build_deck_object()

get_deck_config()

set_deck_config(*config*)

send_deck_cb(*event*)

save_cb(*event*)

load_cb(*event*)

update_deck_graphic_cb(*event*)

start()

AFL.automation.prepare.PrepareWidget.Dropdown

class AFL.automation.prepare.PrepareWidget.**Dropdown**(***kwargs: Any*)

Bases: _Selection

Allows you to select a single item from a dropdown.

Parameters

- **options** (*list*) – The options for the dropdown. This can either be a list of values, e.g. `['Galileo', 'Brahe', 'Hubble']` or `[0, 1, 2]`, a list of (label, value) pairs, e.g. `[('Galileo', 0), ('Brahe', 1), ('Hubble', 2)]`, or a Mapping between labels and values, e.g., `{'Galileo': 0, 'Brahe': 1, 'Hubble': 2}`.
- **index** (*int*) – The index of the current selection.
- **value** (*any*) – The value of the current selection. When programmatically setting the value, a reverse lookup is performed among the options to check that the value is valid. The reverse lookup uses the equality operator by default, but another predicate may be provided via the `equals` keyword argument. For example, when dealing with numpy arrays, one may set `equals=np.array_equal`.
- **label** (*str*) – The label corresponding to the selected value.
- **disabled** (*bool*) – Whether to disable user changes.
- **description** (*str*) – Label for this input group. This should be a string describing the widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.

continues on next page

Table 32 – continued from previous page

get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
index	Selected index
keys	The traits which are synced.
label	Selected label
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
options	Iterable of values, (label, value) pairs, or Mapping between labels and values that the user can select.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Selected value
widget_types	
widgets	

__del__()

Object disposal

__init__(**args*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

index

Selected index

keys

The traits which are synced.

label

Selected label

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

options

Iterable of values, (label, value) pairs, or Mapping between labels and values that the user can select.

The labels are the strings that will be displayed in the UI, representing the actual Python choices, and should be unique.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

style

Styling customizations

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Selected value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.HBox

class AFL.automation.prepare.PrepareWidget.**HBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets horizontally using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Horizontal Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.HBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.

continues on next page

Table 33 – continued from previous page

trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: *Any*

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys:

* owner : the HasTraits instance * old : the old value of the modified trait attribute * new : the new value of the modified trait attribute * name : the name of the modified trait attribute.

- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters
key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters
name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type
The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.Label

class AFL.automation.prepare.PrepareWidget.**Label**(***kwargs: Any*)

Bases: _String

Label widget.

It also renders math inside the string *value* as Latex (requires $ $ or $$ $$ and similar latex tags).

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.

continues on next page

Table 34 – continued from previous page

remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

`style`

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.Layout

class AFL.automation.prepare.PrepareWidget.**Layout**(***kwargs: Any*)

Bases: `Widget`

Layout specification

Defines a layout that can be expressed using CSS. Supports a subset of https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

When a property is also accessible via a shorthand property, we only expose the shorthand.

For example: - `flex-grow`, `flex-shrink` and `flex-basis` are bound to `flex`. - `flex-wrap` and `flex-direction` are bound to `flex-flow`. - `margin-[top/bottom/left/right]` values are bound to `margin`, etc.

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_traits(**traits)	Dynamically add trait attributes to the Widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.

continues on next page

Table 35 – continued from previous page

trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

align_content	The align-content CSS attribute.
align_items	The align-items CSS attribute.
align_self	The align-self CSS attribute.
border	*border* property getter.
border_bottom	The border bottom CSS attribute.
border_left	The border left CSS attribute.
border_right	The border right CSS attribute.
border_top	The border top CSS attribute.
bottom	The bottom CSS attribute.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
display	The display CSS attribute.
flex	The flex CSS attribute.
flex_flow	The flex-flow CSS attribute.
grid_area	The grid-area CSS attribute.
grid_auto_columns	The grid-auto-columns CSS attribute.
grid_auto_flow	The grid-auto-flow CSS attribute.
grid_auto_rows	The grid-auto-rows CSS attribute.
grid_column	The grid-column CSS attribute.
grid_gap	The grid-gap CSS attribute.
grid_row	The grid-row CSS attribute.
grid_template_areas	The grid-template-areas CSS attribute.
grid_template_columns	The grid-template-columns CSS attribute.
grid_template_rows	The grid-template-rows CSS attribute.
height	The height CSS attribute.
justify_content	The justify-content CSS attribute.
justify_items	The justify-items CSS attribute.
keys	The traits which are synced.
left	The left CSS attribute.
log	A trait whose value must be an instance of a specified class.
margin	The margin CSS attribute.
max_height	The max-height CSS attribute.
max_width	The max-width CSS attribute.
min_height	The min-height CSS attribute.
min_width	The min-width CSS attribute.
model_id	Gets the model id of this widget.
object_fit	The object-fit CSS attribute.
object_position	The object-position CSS attribute.
order	The order CSS attribute.
overflow	The overflow CSS attribute.
padding	The padding CSS attribute.

continues on next page

Table 36 – continued from previous page

right	The right CSS attribute.
top	The top CSS attribute.
visibility	The visibility CSS attribute.
widget_types	
widgets	
width	The width CSS attribute.

align_content
The align-content CSS attribute.

align_items
The align-items CSS attribute.

align_self
The align-self CSS attribute.

border_top
The border top CSS attribute.

border_right
The border right CSS attribute.

border_bottom
The border bottom CSS attribute.

border_left
The border left CSS attribute.

bottom
The bottom CSS attribute.

display
The display CSS attribute.

flex
The flex CSS attribute.

flex_flow
The flex-flow CSS attribute.

height
The height CSS attribute.

justify_content
The justify-content CSS attribute.

justify_items
The justify-items CSS attribute.

left
The left CSS attribute.

margin
The margin CSS attribute.

`max_height`
: The max-height CSS attribute.

`max_width`
: The max-width CSS attribute.

`min_height`
: The min-height CSS attribute.

`min_width`
: The min-width CSS attribute.

`overflow`
: The overflow CSS attribute.

`order`
: The order CSS attribute.

`padding`
: The padding CSS attribute.

`right`
: The right CSS attribute.

`top`
: The top CSS attribute.

`visibility`
: The visibility CSS attribute.

`width`
: The width CSS attribute.

`object_fit`
: The object-fit CSS attribute.

`object_position`
: The object-position CSS attribute.

`grid_auto_columns`
: The grid-auto-columns CSS attribute.

`grid_auto_flow`
: The grid-auto-flow CSS attribute.

`grid_auto_rows`
: The grid-auto-rows CSS attribute.

`grid_gap`
: The grid-gap CSS attribute.

`grid_template_rows`
: The grid-template-rows CSS attribute.

`grid_template_columns`
: The grid-template-columns CSS attribute.

grid_template_areas

The grid-template-areas CSS attribute.

grid_row

The grid-row CSS attribute.

grid_column

The grid-column CSS attribute.

grid_area

The grid-area CSS attribute.

__init__(***kwargs*)

Public constructor

property border

border property getter. Return the common value of all side borders if they are identical. Otherwise return None.

__del__()

Object disposal

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.PrepareWidget

class AFL.automation.prepare.PrepareWidget.**PrepareWidget**

Bases: object

__init__()

Methods

SampleSeriesTool_reset_cb(event)
StockBuilder_reset_cb(event)
SweepBuilder_reset_cb(event)
__init__()
start()

__init__()

SweepBuilder_reset_cb(*event*)

StockBuilder_reset_cb(*event*)

SampleSeriesTool_reset_cb(*event*)

start()

AFL.automation.prepare.PrepareWidget.PrepareWidget_Model

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_Model**

Bases: object

__init__()

Methods

__init__()

AFL.automation.prepare.PrepareWidget.PrepareWidget_View

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_View**

Bases: object

__init__()

Methods

__init__()
start(deck_builder_widget)

start(*deck_builder_widget*)

AFL.automation.prepare.PrepareWidget.SampleSeriesWidget

class AFL.automation.prepare.PrepareWidget.**SampleSeriesWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
apply_protocol_order()
apply_protocol_order_cb(event)
build_label(index)
example_label_cb(event)
make_all_labels()
make_all_labels_cb(event)
make_catch_protocol()
make_mixing_wells()
make_protocol()
reset_uuid_cb(event)
start()
submit_cb(event)
submit_mixing_wells_cb(event)
sync_to_prepare_cb(event)
update_mixing_well_preview_cb(event)
update_protocol_order_preview_cb(event)
update_sort_order_cb(event, direction)

__init__(*deck*)

apply_protocol_order_cb(*event*)

apply_protocol_order()

update_protocol_order_preview_cb(*event*)

make_protocol()

make_catch_protocol()

submit_cb(*event*)

build_label(*index*)

example_label_cb(*event*)

make_all_labels_cb(*event*)

make_all_labels()

sync_to_prepare_cb(*event*)

reset_uuid_cb(*event*)

update_sort_order_cb(*event*, *direction*)

make_mixing_wells()

update_mixing_well_preview_cb(*event*)

submit_mixing_wells_cb(*event*)

start()

AFL.automation.prepare.PrepareWidget.StockBuilderWidget

class AFL.automation.prepare.PrepareWidget.**StockBuilderWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
add_stocks_to_deck()
analyze_stocks_cb(event)
get_stock_objects()
get_stock_values()
load_cb(event)
make_stock_cb(event)
remove_stock_cb(event)
save_cb(event[, pkl])
set_units_cb(event, units)
start()
update_location_check_cb(event)

__init__(*deck*)

analyze_stocks_cb(*event*)

get_stock_objects()

add_stocks_to_deck()

get_stock_values()

save_cb(*event*, *pkl=True*)

load_cb(*event*)

update_location_check_cb(*event*)

make_stock_cb(*event*)

remove_stock_cb(*event*)

set_units_cb(*event*, *units*)

start()

AFL.automation.prepare.PrepareWidget.SweepBuilderWidget

class AFL.automation.prepare.PrepareWidget.**SweepBuilderWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
calc_sweep_cb(click)
get_deck()
get_sweep_data()
plot_binary_cb(click)
plot_ternary_cb(click)
start()
update_component_row_cb(event)
validate_sweep_cb(click)

__init__(*deck*)

plot_binary_cb(*click*)

plot_ternary_cb(*click*)

calc_sweep_cb(*click*)

validate_sweep_cb(*click*)

get_sweep_data()

get_deck()

update_component_row_cb(*event*)

start()

AFL.automation.prepare.PrepareWidget.Text

class AFL.automation.prepare.PrepareWidget.**Text**(***kwargs: Any*)

Bases: _String

Single line textbox widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits

continues on next page

Table 37 – continued from previous page

hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_submit(callback[, remove])	(Un)Register a callback to handle text submission.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(**args*, ***kwargs*)

Public constructor

on_submit(*callback*, *remove=False*)

(Un)Register a callback to handle text submission.

Triggered when the user clicks enter.

Parameters

- **callback** (*callable*) – Will be called with exactly one argument: the Widget instance
- **remove** (*bool (optional)*) – Whether to unregister the callback

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.PrepareWidget.VBox

class AFL.automation.prepare.PrepareWidget.**VBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets vertically using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Vertical Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.VBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the frontend.

continues on next page

Table 38 – continued from previous page

send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.

- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

class AFL.automation.prepare.PrepareWidget.**PrepareWidget**

__init__()

SweepBuilder_reset_cb(*event*)

StockBuilder_reset_cb(*event*)

SampleSeriesTool_reset_cb(*event*)

start()

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_Model**

class AFL.automation.prepare.PrepareWidget.**PrepareWidget_View**

start(*deck_builder_widget*)

AFL.automation.prepare.SampleSeriesWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.prepare.SampleSeriesWidget.sqrt

AFL.automation.prepare.SampleSeriesWidget.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
Client([ip, port, username, interactive])	Communicate with APIServer
FloatText(**kwargs)	Displays a float value within a textbox.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
IntText(**kwargs)	Textbox widget that represents an integer.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
SampleSeriesWidget(deck)	
SampleSeriesWidget_Model(deck)	
SampleSeriesWidget_View()	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

AFL.automation.prepare.SampleSeriesWidget.Button

class AFL.automation.prepare.SampleSeriesWidget.**Button**(***kwargs: Any*)

Bases: DOMWidget, CoreWidget

Button widget.

This widget has an *on_click* method that allows you to listen for the user clicking on the button. The click event itself is stateless.

Parameters

- **description** (*str*) – description displayed on the button
- **icon** (*str*) – font-awesome icon names, without the 'fa-' prefix
- **disabled** (*bool*) – whether user interaction is enabled

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.

continues on next page

Table 39 – continued from previous page

click()	Programmatically trigger a click event.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_click(callback[, remove])	Register a callback to execute when the button is clicked.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

button_style	Use a predefined styling for the button.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Button label.
disabled	Enable or disable user changes.
icon	Font-awesome icon names, without the 'fa-' prefix.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

description

Button label.

disabled

Enable or disable user changes.

icon

Font-awesome icon names, without the 'fa-' prefix.

button_style

Use a predefined styling for the button.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(***kwargs*)

Public constructor

on_click(*callback*, *remove=False*)

Register a callback to execute when the button is clicked.

The callback will be called with one argument, the clicked button widget instance.

Parameters
remove (*bool (optional)*) – Set to true to remove the callback from the list of callbacks.

click()

Programmatically trigger a click event.

This will call the callbacks registered to the clicked button widget instance.

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(**traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.Checkbox

class AFL.automation.prepare.SampleSeriesWidget.**Checkbox**(***kwargs: Any*)

Bases: _Bool

Displays a boolean *value* in the form of a checkbox.

Parameters

- **value** (*{True,False}*) – value of the checkbox: True-checked, False-unchecked
- **description** (*str*) – description displayed next to the checkbox
- **indent** (*{True,False}*) – indent the control to align with other controls with a description. The style.description_width attribute controls this width for consistence with other controls.

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.

continues on next page

Table 40 – continued from previous page

get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes.
indent	Indent the control to align with other controls with a description.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Bool value
widget_types	
widgets	

indent

Indent the control to align with other controls with a description.

style

Styling customizations

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: *Any*

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(****metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(****metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(****metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Bool value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.Client

class AFL.automation.prepare.SampleSeriesWidget.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	

continues on next page

Table 41 – continued from previous page

get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.prepare.SampleSeriesWidget.FloatText

class AFL.automation.prepare.SampleSeriesWidget.**FloatText**(***kwargs: Any*)

Bases: _Float

Displays a float value within a textbox. For a textbox in which the value must be within a specific range, use BoundedFloatText.

Parameters

- **value** (*float*) – value displayed
- **step** (*float*) – step of the increment (if None, any step is allowed)
- **description** (*str*) – description displayed next to the text box

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.

continues on next page

Table 42 – continued from previous page

on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
step	Minimum step to increment the value
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Float value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

step

Minimum step to increment the value

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

style

Styling customizations

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: *Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Float value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.HBox

class AFL.automation.prepare.SampleSeriesWidget.**HBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets horizontally using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Horizontal Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.HBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits

continues on next page

Table 43 – continued from previous page

hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False,

then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.IntText

class AFL.automation.prepare.SampleSeriesWidget.**IntText**(***kwargs: Any*)

Bases: _Int

Textbox widget that represents an integer.

__init__(*value=None*, ***kwargs*)

Parameters
value (*integer*) – The initial value.

Methods

__init__([value])	
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.

continues on next page

Table 44 – continued from previous page

get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
step	Minimum step to increment the value
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Int value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

step

Minimum step to increment the value

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Parameters
value (*integer*) – The initial value.

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

`get_interact_value`()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

`static get_manager_state`(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

`get_state`(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

`get_view_spec`()

`static handle_comm_opened`(*comm*, *msg*)

Static method, called when a widget is constructed.

`classmethod handle_control_comm_opened`(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

`has_trait`(*name: str*) → bool

Returns True if the object has a trait with the specified name.

`hold_sync`()

Hold syncing any state until the outermost context manager exits

`hold_trait_notifications`() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

`keys`

The traits which are synced.

`layout`

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

style

Styling customizations

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Int value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.Label

class AFL.automation.prepare.SampleSeriesWidget.**Label**(***kwargs: Any*)

Bases: _String

Label widget.

It also renders math inside the string *value* as Latex (requires $ $ or $$ $$ and similar latex tags).

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.

continues on next page

Table 45 – continued from previous page

close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

`get_state`(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

`get_view_spec`()

`static handle_comm_opened`(*comm*, *msg*)

Static method, called when a widget is constructed.

`classmethod handle_control_comm_opened`(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

`has_trait`(*name: str*) → bool

Returns True if the object has a trait with the specified name.

`hold_sync`()

Hold syncing any state until the outermost context manager exits

`hold_trait_notifications`() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

`keys`

The traits which are synced.

`layout`

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

`log`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

`property model_id`

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

`notify_change`(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(****metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(****metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(****metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.Layout

class AFL.automation.prepare.SampleSeriesWidget.**Layout**(***kwargs: Any*)

Bases: Widget

Layout specification

Defines a layout that can be expressed using CSS. Supports a subset of https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

When a property is also accessible via a shorthand property, we only expose the shorthand.

For example: - `flex-grow`, `flex-shrink` and `flex-basis` are bound to `flex`. - `flex-wrap` and `flex-direction` are bound to `flex-flow`. - `margin-[top/bottom/left/right]` values are bound to `margin`, etc.

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_traits(**traits)	Dynamically add trait attributes to the Widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.

continues on next page

Table 46 – continued from previous page

handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

align_content	The align-content CSS attribute.
align_items	The align-items CSS attribute.
align_self	The align-self CSS attribute.
border	*border* property getter.
border_bottom	The border bottom CSS attribute.
border_left	The border left CSS attribute.
border_right	The border right CSS attribute.
border_top	The border top CSS attribute.
bottom	The bottom CSS attribute.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
display	The display CSS attribute.

continues on next page

Table 47 – continued from previous page

flex	The flex CSS attribute.
flex_flow	The flex-flow CSS attribute.
grid_area	The grid-area CSS attribute.
grid_auto_columns	The grid-auto-columns CSS attribute.
grid_auto_flow	The grid-auto-flow CSS attribute.
grid_auto_rows	The grid-auto-rows CSS attribute.
grid_column	The grid-column CSS attribute.
grid_gap	The grid-gap CSS attribute.
grid_row	The grid-row CSS attribute.
grid_template_areas	The grid-template-areas CSS attribute.
grid_template_columns	The grid-template-columns CSS attribute.
grid_template_rows	The grid-template-rows CSS attribute.
height	The height CSS attribute.
justify_content	The justify-content CSS attribute.
justify_items	The justify-items CSS attribute.
keys	The traits which are synced.
left	The left CSS attribute.
log	A trait whose value must be an instance of a specified class.
margin	The margin CSS attribute.
max_height	The max-height CSS attribute.
max_width	The max-width CSS attribute.
min_height	The min-height CSS attribute.
min_width	The min-width CSS attribute.
model_id	Gets the model id of this widget.
object_fit	The object-fit CSS attribute.
object_position	The object-position CSS attribute.
order	The order CSS attribute.
overflow	The overflow CSS attribute.
padding	The padding CSS attribute.
right	The right CSS attribute.
top	The top CSS attribute.
visibility	The visibility CSS attribute.
widget_types	
widgets	
width	The width CSS attribute.

align_content

The align-content CSS attribute.

align_items

The align-items CSS attribute.

align_self

The align-self CSS attribute.

border_top

The border top CSS attribute.

border_right

The border right CSS attribute.

border_bottom
The border bottom CSS attribute.

border_left
The border left CSS attribute.

bottom
The bottom CSS attribute.

display
The display CSS attribute.

flex
The flex CSS attribute.

flex_flow
The flex-flow CSS attribute.

height
The height CSS attribute.

justify_content
The justify-content CSS attribute.

justify_items
The justify-items CSS attribute.

left
The left CSS attribute.

margin
The margin CSS attribute.

max_height
The max-height CSS attribute.

max_width
The max-width CSS attribute.

min_height
The min-height CSS attribute.

min_width
The min-width CSS attribute.

overflow
The overflow CSS attribute.

order
The order CSS attribute.

padding
The padding CSS attribute.

right
The right CSS attribute.

top

The top CSS attribute.

visibility

The visibility CSS attribute.

width

The width CSS attribute.

object_fit

The object-fit CSS attribute.

object_position

The object-position CSS attribute.

grid_auto_columns

The grid-auto-columns CSS attribute.

grid_auto_flow

The grid-auto-flow CSS attribute.

grid_auto_rows

The grid-auto-rows CSS attribute.

grid_gap

The grid-gap CSS attribute.

grid_template_rows

The grid-template-rows CSS attribute.

grid_template_columns

The grid-template-columns CSS attribute.

grid_template_areas

The grid-template-areas CSS attribute.

grid_row

The grid-row CSS attribute.

grid_column

The grid-column CSS attribute.

grid_area

The grid-area CSS attribute.

__init__(***kwargs*)

Public constructor

property border

border property getter. Return the common value of all side borders if they are identical. Otherwise return None.

__del__()

Object disposal

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

`get_view_spec()`

`static handle_comm_opened(comm, msg)`

Static method, called when a widget is constructed.

`classmethod handle_control_comm_opened(comm, msg)`

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

`has_trait(name: str) → bool`

Returns True if the object has a trait with the specified name.

`hold_sync()`

Hold syncing any state until the outermost context manager exits

`hold_trait_notifications() → Any`

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

`keys`

The traits which are synced.

`log`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

`property model_id`

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

`notify_change(change)`

Called when a property has changed.

`observe(handler: Callable[[...], Any], names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All, type: Sentinel | str = 'change') → None`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.

- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters
key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters
name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type
The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False,

then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.SampleSeriesWidget

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
apply_protocol_order()
apply_protocol_order_cb(event)
build_label(index)
example_label_cb(event)
make_all_labels()
make_all_labels_cb(event)
make_catch_protocol()
make_mixing_wells()
make_protocol()
reset_uuid_cb(event)
start()
submit_cb(event)
submit_mixing_wells_cb(event)
sync_to_prepare_cb(event)
update_mixing_well_preview_cb(event)
update_protocol_order_preview_cb(event)
update_sort_order_cb(event, direction)

__init__(*deck*)

apply_protocol_order_cb(*event*)

apply_protocol_order()

update_protocol_order_preview_cb(*event*)

make_protocol()

make_catch_protocol()

submit_cb(*event*)

build_label(*index*)

example_label_cb(*event*)

make_all_labels_cb(*event*)

make_all_labels()

sync_to_prepare_cb(*event*)

reset_uuid_cb(*event*)

update_sort_order_cb(*event*, *direction*)

make_mixing_wells()

update_mixing_well_preview_cb(*event*)

submit_mixing_wells_cb(*event*)

start()

AFL.automation.prepare.SampleSeriesWidget.SampleSeriesWidget_Model

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_Model**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)	
adjust_protocol_order(mix_order)	

__init__(*deck*)

adjust_protocol_order(*mix_order*)

AFL.automation.prepare.SampleSeriesWidget.SampleSeriesWidget_View

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_View**

Bases: object

__init__()

Methods

__init__()	
make_component_grid(components, nsamples)	
make_mixing_wells()	
make_pipette_params(name)	
start(components, nsamples, stock_names)	

__init__()

make_component_grid(*components*, *nsamples*)

make_pipette_params(*name*)

make_mixing_wells()

start(*components*, *nsamples*, *stock_names*)

AFL.automation.prepare.SampleSeriesWidget.Text

class AFL.automation.prepare.SampleSeriesWidget.**Text**(***kwargs: Any*)

Bases: _String

Single line textbox widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding

continues on next page

Table 48 – continued from previous page

get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_submit(callback[, remove])	(Un)Register a callback to handle text submission.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(**args*, ***kwargs*)

Public constructor

on_submit(*callback*, *remove=False*)

(Un)Register a callback to handle text submission.

Triggered when the user clicks enter.

Parameters

- **callback** (*callable*) – Will be called with exactly one argument: the Widget instance
- **remove** (*bool (optional)*) – Whether to unregister the callback

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SampleSeriesWidget.VBox

class AFL.automation.prepare.SampleSeriesWidget.**VBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets vertically using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Vertical Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.VBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the frontend.

continues on next page

Table 49 – continued from previous page

send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.

- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget**(*deck*)

__init__(*deck*)

apply_protocol_order_cb(*event*)

apply_protocol_order()

update_protocol_order_preview_cb(*event*)

make_protocol()

make_catch_protocol()

submit_cb(*event*)

build_label(*index*)

example_label_cb(*event*)

make_all_labels_cb(*event*)

make_all_labels()

sync_to_prepare_cb(*event*)

reset_uuid_cb(*event*)

update_sort_order_cb(*event*, *direction*)

make_mixing_wells()

update_mixing_well_preview_cb(*event*)

submit_mixing_wells_cb(*event*)

start()

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_Model**(*deck*)

__init__(*deck*)

adjust_protocol_order(*mix_order*)

class AFL.automation.prepare.SampleSeriesWidget.**SampleSeriesWidget_View**

__init__()

make_component_grid(*components*, *nsamples*)

make_pipette_params(*name*)

make_mixing_wells()

start(*components*, *nsamples*, *stock_names*)

AFL.automation.prepare.StockBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.prepare.StockBuilderWidget.sqrt

AFL.automation.prepare.StockBuilderWidget.**sqrt**(*x*, */*)

Return the square root of x.

Classes

StockBuilderWidget(deck)
StockBuilderWidget_Model(deck)
StockBuilderWidget_View()

AFL.automation.prepare.StockBuilderWidget.StockBuilderWidget

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
add_stocks_to_deck()
analyze_stocks_cb(event)
get_stock_objects()
get_stock_values()
load_cb(event)
make_stock_cb(event)
remove_stock_cb(event)
save_cb(event[, pkl])
set_units_cb(event, units)
start()
update_location_check_cb(event)

__init__(*deck*)

analyze_stocks_cb(*event*)

get_stock_objects()

add_stocks_to_deck()

get_stock_values()

save_cb(*event*, *pkl=True*)

load_cb(*event*)

update_location_check_cb(*event*)

make_stock_cb(*event*)

remove_stock_cb(*event*)

set_units_cb(*event*, *units*)

start()

AFL.automation.prepare.StockBuilderWidget.StockBuilderWidget_Model

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_Model**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)
add_stocks_to_deck(all_stocks_dict)
to_stock_objects(all_stocks_dict)

__init__(*deck*)

add_stocks_to_deck(*all_stocks_dict*)

to_stock_objects(*all_stocks_dict*)

AFL.automation.prepare.StockBuilderWidget.StockBuilderWidget_View

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_View**

Bases: object

__init__()

Methods

__init__()
make_stock_tab(stock_name, components)
start()

__init__()

make_stock_tab(*stock_name*, *components*)

start()

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget**(*deck*)

__init__(*deck*)

analyze_stocks_cb(*event*)

get_stock_objects()

add_stocks_to_deck()

get_stock_values()

save_cb(*event*, *pkl=True*)

load_cb(*event*)

update_location_check_cb(*event*)

make_stock_cb(*event*)

remove_stock_cb(*event*)

set_units_cb(*event*, *units*)

start()

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_Model**(*deck*)

__init__(*deck*)

add_stocks_to_deck(*all_stocks_dict*)

to_stock_objects(*all_stocks_dict*)

class AFL.automation.prepare.StockBuilderWidget.**StockBuilderWidget_View**

__init__()

make_stock_tab(*stock_name*, *components*)

start()

AFL.automation.prepare.SweepBuilderWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.prepare.SweepBuilderWidget.sqrt

AFL.automation.prepare.SweepBuilderWidget.**sqrt**(*x*, */*)

Return the square root of x.

Classes

Button(**kwargs)	Button widget.
Checkbox(**kwargs)	Displays a boolean *value* in the form of a checkbox.
HBox(**kwargs)	Displays multiple widgets horizontally using the flexible box model.
Label(**kwargs)	Label widget.
Layout(**kwargs)	Layout specification
SweepBuilderWidget(deck)	
SweepBuilderWidget_Model(deck)	
SweepBuilderWidget_View()	
Text(**kwargs)	Single line textbox widget.
VBox(**kwargs)	Displays multiple widgets vertically using the flexible box model.

AFL.automation.prepare.SweepBuilderWidget.Button

class AFL.automation.prepare.SweepBuilderWidget.**Button**(***kwargs: Any*)

Bases: DOMWidget, CoreWidget

Button widget.

This widget has an *on_click* method that allows you to listen for the user clicking on the button. The click event itself is stateless.

Parameters

- **description** (*str*) – description displayed on the button
- **icon** (*str*) – font-awesome icon names, without the 'fa-' prefix
- **disabled** (*bool*) – whether user interaction is enabled

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
click()	Programmatically trigger a click event.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.

continues on next page

Table 50 – continued from previous page

on_click(callback[, remove])	Register a callback to execute when the button is clicked.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

button_style	Use a predefined styling for the button.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Button label.
disabled	Enable or disable user changes.
icon	Font-awesome icon names, without the 'fa-' prefix.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

description

Button label.

disabled

Enable or disable user changes.

icon

Font-awesome icon names, without the 'fa-' prefix.

button_style

Use a predefined styling for the button.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(***kwargs*)

Public constructor

on_click(*callback*, *remove=False*)

Register a callback to execute when the button is clicked.

The callback will be called with one argument, the clicked button widget instance.

Parameters

remove (*bool (optional)*) – Set to true to remove the callback from the list of callbacks.

click()

Programmatically trigger a click event.

This will call the callbacks registered to the clicked button widget instance.

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: *Any*

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.

- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters
key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters
name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type
The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: *Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.Checkbox

class AFL.automation.prepare.SweepBuilderWidget.**Checkbox**(***kwargs: Any*)

Bases: _Bool

Displays a boolean *value* in the form of a checkbox.

Parameters

- **value** (*{True,False}*) – value of the checkbox: True-checked, False-unchecked
- **description** (*str*) – description displayed next to the checkbox
- **indent** (*{True,False}*) – indent the control to align with other controls with a description. The style.description_width attribute controls this width for consistence with other controls.

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a dict of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.

continues on next page

Table 51 – continued from previous page

on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes.
indent	Indent the control to align with other controls with a description.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
style	Styling customizations
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	Bool value
widget_types	
widgets	

indent

Indent the control to align with other controls with a description.

style

Styling customizations

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

disabled

Enable or disable user changes.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(**kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

Bool value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.HBox

class AFL.automation.prepare.SweepBuilderWidget.**HBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets horizontally using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Horizontal Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.HBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.

continues on next page

Table 52 – continued from previous page

open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(**traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(****metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(****metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(****metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.Label

class AFL.automation.prepare.SweepBuilderWidget.**Label**(***kwargs: Any*)

Bases: _String

Label widget.

It also renders math inside the string *value* as Latex (requires $ $ or $$ $$ and similar latex tags).

__init__(*value=None*, ***kwargs*)

Public constructor

Methods

__init__([value])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

continues on next page

Table 53 – continued from previous page

has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__del__()

Object disposal

__init__(*value=None*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: *Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

`get_state`(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

`get_view_spec`()

`static handle_comm_opened`(*comm*, *msg*)

Static method, called when a widget is constructed.

`classmethod handle_control_comm_opened`(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

`has_trait`(*name: str*) → bool

Returns True if the object has a trait with the specified name.

`hold_sync`()

Hold syncing any state until the outermost context manager exits

`hold_trait_notifications`() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

`keys`

The traits which are synced.

`layout`

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

`log`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

`property model_id`

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

`notify_change`(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.

- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.Layout

class AFL.automation.prepare.SweepBuilderWidget.**Layout**(***kwargs: Any*)

Bases: Widget

Layout specification

Defines a layout that can be expressed using CSS. Supports a subset of https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

When a property is also accessible via a shorthand property, we only expose the shorthand.

For example: - `flex-grow`, `flex-shrink` and `flex-basis` are bound to `flex`. - `flex-wrap` and `flex-direction` are bound to `flex-flow`. - `margin-[top/bottom/left/right]` values are bound to `margin`, etc.

__init__(***kwargs*)

Public constructor

Methods

__init__(**kwargs)	Public constructor
add_traits(**traits)	Dynamically add trait attributes to the Widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.

continues on next page

Table 54 – continued from previous page

handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.
traits(**metadata)	Get a `dict` of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

align_content	The align-content CSS attribute.
align_items	The align-items CSS attribute.
align_self	The align-self CSS attribute.
border	*border* property getter.
border_bottom	The border bottom CSS attribute.
border_left	The border left CSS attribute.
border_right	The border right CSS attribute.
border_top	The border top CSS attribute.
bottom	The bottom CSS attribute.
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
display	The display CSS attribute.

continues on next page

Table 55 – continued from previous page

flex	The flex CSS attribute.
flex_flow	The flex-flow CSS attribute.
grid_area	The grid-area CSS attribute.
grid_auto_columns	The grid-auto-columns CSS attribute.
grid_auto_flow	The grid-auto-flow CSS attribute.
grid_auto_rows	The grid-auto-rows CSS attribute.
grid_column	The grid-column CSS attribute.
grid_gap	The grid-gap CSS attribute.
grid_row	The grid-row CSS attribute.
grid_template_areas	The grid-template-areas CSS attribute.
grid_template_columns	The grid-template-columns CSS attribute.
grid_template_rows	The grid-template-rows CSS attribute.
height	The height CSS attribute.
justify_content	The justify-content CSS attribute.
justify_items	The justify-items CSS attribute.
keys	The traits which are synced.
left	The left CSS attribute.
log	A trait whose value must be an instance of a specified class.
margin	The margin CSS attribute.
max_height	The max-height CSS attribute.
max_width	The max-width CSS attribute.
min_height	The min-height CSS attribute.
min_width	The min-width CSS attribute.
model_id	Gets the model id of this widget.
object_fit	The object-fit CSS attribute.
object_position	The object-position CSS attribute.
order	The order CSS attribute.
overflow	The overflow CSS attribute.
padding	The padding CSS attribute.
right	The right CSS attribute.
top	The top CSS attribute.
visibility	The visibility CSS attribute.
widget_types	
widgets	
width	The width CSS attribute.

align_content

The align-content CSS attribute.

align_items

The align-items CSS attribute.

align_self

The align-self CSS attribute.

border_top

The border top CSS attribute.

border_right

The border right CSS attribute.

`border_bottom`
: The border bottom CSS attribute.

`border_left`
: The border left CSS attribute.

`bottom`
: The bottom CSS attribute.

`display`
: The display CSS attribute.

`flex`
: The flex CSS attribute.

`flex_flow`
: The flex-flow CSS attribute.

`height`
: The height CSS attribute.

`justify_content`
: The justify-content CSS attribute.

`justify_items`
: The justify-items CSS attribute.

`left`
: The left CSS attribute.

`margin`
: The margin CSS attribute.

`max_height`
: The max-height CSS attribute.

`max_width`
: The max-width CSS attribute.

`min_height`
: The min-height CSS attribute.

`min_width`
: The min-width CSS attribute.

`overflow`
: The overflow CSS attribute.

`order`
: The order CSS attribute.

`padding`
: The padding CSS attribute.

`right`
: The right CSS attribute.

top

The top CSS attribute.

visibility

The visibility CSS attribute.

width

The width CSS attribute.

object_fit

The object-fit CSS attribute.

object_position

The object-position CSS attribute.

grid_auto_columns

The grid-auto-columns CSS attribute.

grid_auto_flow

The grid-auto-flow CSS attribute.

grid_auto_rows

The grid-auto-rows CSS attribute.

grid_gap

The grid-gap CSS attribute.

grid_template_rows

The grid-template-rows CSS attribute.

grid_template_columns

The grid-template-columns CSS attribute.

grid_template_areas

The grid-template-areas CSS attribute.

grid_row

The grid-row CSS attribute.

grid_column

The grid-column CSS attribute.

grid_area

The grid-area CSS attribute.

__init__(***kwargs*)

Public constructor

property border

border property getter. Return the common value of all side borders if they are identical. Otherwise return None.

__del__()

Object disposal

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

`get_view_spec()`

`static handle_comm_opened(comm, msg)`

Static method, called when a widget is constructed.

`classmethod handle_control_comm_opened(comm, msg)`

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

`has_trait(name: str) → bool`

Returns True if the object has a trait with the specified name.

`hold_sync()`

Hold syncing any state until the outermost context manager exits

`hold_trait_notifications() → Any`

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

`keys`

The traits which are synced.

`log`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

`property model_id`

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

`notify_change(change)`

Called when a property has changed.

`observe(handler: Callable[[...], Any], names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All, type: Sentinel | str = 'change') → None`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.

- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters
key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters
name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type
The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(***metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False,

then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.SweepBuilderWidget

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)	
calc_sweep_cb(click)	
get_deck()	
get_sweep_data()	
plot_binary_cb(click)	
plot_ternary_cb(click)	
start()	
update_component_row_cb(event)	
validate_sweep_cb(click)	

__init__(*deck*)

plot_binary_cb(*click*)

plot_ternary_cb(*click*)

calc_sweep_cb(*click*)

validate_sweep_cb(*click*)

get_sweep_data()

get_deck()

update_component_row_cb(*event*)

start()

AFL.automation.prepare.SweepBuilderWidget.SweepBuilderWidget_Model

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_Model**(*deck*)

Bases: object

__init__(*deck*)

Methods

__init__(deck)	
calc_sweep(sweep_dict, progress)	
validate_sweep(tolerance[, progress])	

__init__(*deck*)

validate_sweep(*tolerance*, *progress=None*)

calc_sweep(*sweep_dict*, *progress*)

AFL.automation.prepare.SweepBuilderWidget.SweepBuilderWidget_View

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_View**

Bases: object

__init__()

Methods

__init__()	
make_binary_plot(component_names)	
make_stock_grid(component_names)	
make_ternary_plot(component_names)	
start(component_names)	

__init__()

make_stock_grid(*component_names*)

make_ternary_plot(*component_names*)

make_binary_plot(*component_names*)

start(*component_names*)

AFL.automation.prepare.SweepBuilderWidget.Text

class AFL.automation.prepare.SweepBuilderWidget.**Text**(***kwargs: Any*)

Bases: _String

Single line textbox widget.

__init__(**args*, ***kwargs*)

Public constructor

Methods

__init__(*args, **kwargs)	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.

continues on next page

Table 56 – continued from previous page

class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_interact_value()	Return the value for this widget which should be passed to interactive functions.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_submit(callback[, remove])	(Un)Register a callback to handle text submission.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the front-end.
send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a `dict` of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A `dict` of trait names and their values.

continues on next page

Table 56 – continued from previous page

traits(**metadata)	Get a dict of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

comm	A trait which allows any value.
continuous_update	Update the value as the user types.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
description	Description of the control.
description_allow_html	Accept HTML in the description.
description_tooltip	The tooltip information. .. deprecated :: 8.0.0 Use tooltip attribute instead.
disabled	Enable or disable user changes
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
placeholder	Placeholder text to display when nothing has been typed
style	An instance trait which coerces a dict to an instance.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
value	String value
widget_types	
widgets	

disabled

Enable or disable user changes

continuous_update

Update the value as the user types. If False, update on submission, e.g., pressing Enter or navigating away.

style

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

__init__(**args*, ***kwargs*)

Public constructor

on_submit(*callback*, *remove=False*)

(Un)Register a callback to handle text submission.

Triggered when the user clicks enter.

Parameters

- **callback** (*callable*) – Will be called with exactly one argument: the Widget instance
- **remove** (*bool (optional)*) – Whether to unregister the callback

__del__()

Object disposal

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(***metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

description

Description of the control.

description_allow_html

Accept HTML in the description.

property description_tooltip

The tooltip information. .. deprecated :: 8.0.0

Use tooltip attribute instead.

focus()

Focus on the widget.

get_interact_value()

Return the value for this widget which should be passed to interactive functions. Custom widgets can change this method to process the raw value `self.value`.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

placeholder

Placeholder text to display when nothing has been typed

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(**kwargs: *Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(*names: *str*, **metadata: *Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: *str* | *None* = *None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: *str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: *str*, *key: *str*, *default: *Any* = *None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: *Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: *Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type

A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(***metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

value

String value

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

AFL.automation.prepare.SweepBuilderWidget.VBox

class AFL.automation.prepare.SweepBuilderWidget.**VBox**(***kwargs: Any*)

Bases: Box

Displays multiple widgets vertically using the flexible box model.

Parameters

- **children** (*iterable of Widget instances*) – list of widgets to display
- **box_style** (*str*) – one of 'success', 'info', 'warning' or 'danger', or ''. Applies a predefined style to the box. Defaults to '', which applies no pre-defined style.

Examples

```
>>> import ipywidgets as widgets
>>> title_widget = widgets.HTML('<em>Vertical Box Example</em>')
>>> slider = widgets.IntSlider()
>>> widgets.VBox([title_widget, slider])
```

__init__(*children=()*, ***kwargs*)

Public constructor

Methods

__init__([children])	Public constructor
add_class(className)	Adds a class to the top level element of the widget.
add_traits(**traits)	Dynamically add trait attributes to the Widget.
blur()	Blur the widget.
class_own_trait_events(name)	Get a dict of all event handlers defined on this class, not a parent.
class_own_traits(**metadata)	Get a dict of all the traitlets defined on this class, not a parent.
class_trait_names(**metadata)	Get a list of all the names of this class' traits.
class_traits(**metadata)	Get a `dict` of all the traits of this class.
close()	Close method.
close_all()	
focus()	Focus on the widget.
get_manager_state([drop_defaults, widgets])	Returns the full state for a widget manager for embedding
get_state([key, drop_defaults])	Gets the widget state, or a piece of it.
get_view_spec()	
handle_comm_opened(comm, msg)	Static method, called when a widget is constructed.
handle_control_comm_opened(comm, msg)	Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received
has_trait(name)	Returns True if the object has a trait with the specified name.
hold_sync()	Hold syncing any state until the outermost context manager exits
hold_trait_notifications()	Context manager for bundling trait change notifications and cross validation.
notify_change(change)	Called when a property has changed.
observe(handler[, names, type])	Setup a handler to be called when a trait changes.
on_msg(callback[, remove])	(Un)Register a custom msg receive callback.
on_trait_change([handler, name, remove])	DEPRECATED: Setup a handler to be called when a trait changes.
on_widget_constructed(callback)	Registers a callback to be called when a widget is constructed.
open()	Open a comm to the frontend if one isn't already open.
remove_class(className)	Removes a class from the top level element of the widget.
send(content[, buffers])	Sends a custom msg to the widget model in the frontend.

continues on next page

Table 57 – continued from previous page

send_state([key])	Sends the widget state, or a piece of it, to the front-end, if it exists.
set_state(sync_data)	Called when a state is received from the front-end.
set_trait(name, value)	Forcibly sets trait attribute, including read-only attributes.
setup_instance(**kwargs)	This is called **before** self.__init__ is called.
trait_defaults(*names, **metadata)	Return a trait's default value or a dictionary of them
trait_events([name])	Get a dict of all the event handlers of this class.
trait_has_value(name)	Returns True if the specified trait has a value.
trait_metadata(traitname, key[, default])	Get metadata values for trait by key.
trait_names(**metadata)	Get a list of all the names of this class' traits.
trait_values(**metadata)	A dict of trait names and their values.
traits(**metadata)	Get a dict of all the traits of this class.
unobserve(handler[, names, type])	Remove a trait change handler.
unobserve_all([name])	Remove trait change handlers of any type for the specified name.

Attributes

box_style	Use a predefined styling for the box.
children	List of widget children
comm	A trait which allows any value.
cross_validation_lock	A contextmanager for running a block with our cross validation lock set to True.
keys	The traits which are synced.
layout	An instance trait which coerces a dict to an instance.
log	A trait whose value must be an instance of a specified class.
model_id	Gets the model id of this widget.
tabbable	Is widget tabbable?
tooltip	A tooltip caption.
widget_types	
widgets	

__del__()

Object disposal

__init__(*children=()*, ***kwargs*)

Public constructor

add_class(*className*)

Adds a class to the top level element of the widget.

Doesn't add the class if it already exists.

add_traits(***traits*)

Dynamically add trait attributes to the Widget.

blur()

Blur the widget.

box_style

Use a predefined styling for the box.

children

List of widget children

classmethod class_own_trait_events(*name: str*) → dict[str, EventHandler]

Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

classmethod class_traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

close()

Close method.

Closes the underlying comm. When the comm is closed, all of the widget views are automatically removed from the front-end.

classmethod close_all()

comm

A trait which allows any value.

property cross_validation_lock: Any

A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

focus()

Focus on the widget.

static get_manager_state(*drop_defaults=False*, *widgets=None*)

Returns the full state for a widget manager for embedding

Parameters

- **drop_defaults** – when True, it will not include default value
- **widgets** – list with widgets to include in the state (or all widgets when None)

Returns

get_state(*key=None*, *drop_defaults=False*)

Gets the widget state, or a piece of it.

Parameters

key (*unicode or iterable (optional)*) – A single property's name or iterable of property names to get.

Returns

- **state** (*dict of states*)
- **metadata** (*dict*) – metadata for each field: {key: metadata}

get_view_spec()

static handle_comm_opened(*comm*, *msg*)

Static method, called when a widget is constructed.

classmethod handle_control_comm_opened(*comm*, *msg*)

Class method, called when the comm-open message on the "jupyter.widget.control" comm channel is received

has_trait(*name: str*) → bool

Returns True if the object has a trait with the specified name.

hold_sync()

Hold syncing any state until the outermost context manager exits

hold_trait_notifications() → Any

Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

keys

The traits which are synced.

layout

An instance trait which coerces a dict to an instance.

This lets the instance be specified as a dict, which is used to initialize the instance.

Also, we default to a trivial instance, even if args and kwargs is not specified.

log

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

property model_id

Gets the model id of this widget.

If a Comm doesn't exist yet, a Comm will be created automagically.

notify_change(*change*)

Called when a property has changed.

observe(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_msg(*callback*, *remove=False*)

(Un)Register a custom msg receive callback.

Parameters

- **callback** (*callable*) – callback will be passed three arguments when a message arrives:

```
callback(widget, content, buffers)
```

- **remove** (*bool*) – True if the callback should be unregistered.

on_trait_change(*handler: EventHandler | None = None*, *name: Sentinel | str | None = None*, *remove: bool = False*) → None

DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

static on_widget_constructed(*callback*)

Registers a callback to be called when a widget is constructed.

The callback must have the following signature: callback(widget)

open()

Open a comm to the frontend if one isn't already open.

remove_class(*className*)

Removes a class from the top level element of the widget.

Doesn't remove the class if it doesn't exist.

send(*content*, *buffers=None*)

Sends a custom msg to the widget model in the front-end.

Parameters

- **content** (*dict*) – Content of the message to send.
- **buffers** (*list of binary buffers*) – Binary buffers to send with message

send_state(*key=None*)

Sends the widget state, or a piece of it, to the front-end, if it exists.

Parameters

key (*unicode, or iterable (optional)*) – A single property's name or iterable of property names to sync with the front-end.

set_state(*sync_data*)

Called when a state is received from the front-end.

set_trait(*name: str*, *value: Any*) → None

Forcibly sets trait attribute, including read-only attributes.

setup_instance(***kwargs: Any*) → None

This is called **before** self.__init__ is called.

tabbable

Is widget tabbable?

tooltip

A tooltip caption.

trait_defaults(**names: str*, ***metadata: Any*) → dict[str, Any] | Sentinel

Return a trait's default value or a dictionary of them

Notes

Dynamically generated default values may depend on the current state of the object.

classmethod trait_events(*name: str | None = None*) → dict[str, EventHandler]

Get a `dict` of all the event handlers of this class.

Parameters

name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Return type

The event handlers associated with a trait name, or all event handlers.

trait_has_value(*name: str*) → bool

Returns True if the specified trait has a value.

This will return false even if `getattr` would return a dynamically generated default value. These default values will be recognized as existing only after they have been generated.

Example

```
class MyClass(HasTraits):
    i = Int()


mc = MyClass()
assert not mc.trait_has_value("i")
mc.i  # generates a default value
assert mc.trait_has_value("i")
```

trait_metadata(*traitname: str*, *key: str*, *default: Any = None*) → Any

Get metadata values for trait by key.

trait_names(**metadata: Any*) → list[str]

Get a list of all the names of this class' traits.

trait_values(**metadata: Any*) → dict[str, Any]

A `dict` of trait names and their values.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

Return type
A `dict` of trait names and their values.

Notes

Trait values are retrieved via `getattr`, any exceptions raised by traits or the operations they may trigger will result in the absence of a trait value in the result `dict`.

traits(**metadata: Any*) → dict[str, TraitType[Any, Any]]

Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler: Callable[[...], Any]*, *names: Sentinel | str | Iterable[Sentinel | str] = traitlets.All*, *type: Sentinel | str = 'change'*) → None

Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.

- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name: str | Any = traitlets.All*) → None

Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

widget_types = <ipywidgets.widgets.widget.WidgetRegistry object>

widgets = {}

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget**(*deck*)

__init__(*deck*)

plot_binary_cb(*click*)

plot_ternary_cb(*click*)

calc_sweep_cb(*click*)

validate_sweep_cb(*click*)

get_sweep_data()

get_deck()

update_component_row_cb(*event*)

start()

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_Model**(*deck*)

__init__(*deck*)

validate_sweep(*tolerance*, *progress=None*)

calc_sweep(*sweep_dict*, *progress*)

class AFL.automation.prepare.SweepBuilderWidget.**SweepBuilderWidget_View**

__init__()

make_stock_grid(*component_names*)

make_ternary_plot(*component_names*)

make_binary_plot(*component_names*)

start(*component_names*)

AFL.automation.prepare.factory

Functions

HD2OFactory(name[, phi_D2O, sld, properties])	Create a list of H2O/D2O solutions
compositionSweepFactory(name, components, ...)	
has_units(value)	
is_concentration(value)	
is_mass(value)	
is_molarity(value)	
is_volume(value)	
listify(obj)	

AFL.automation.prepare.factory.HD2OFactory

AFL.automation.prepare.factory.**HD2OFactory**(*name*, *phi_D2O=None*, *sld=None*, *properties=None*)

Create a list of H2O/D2O solutions

AFL.automation.prepare.factory.compositionSweepFactory

AFL.automation.prepare.factory.**compositionSweepFactory**(*name*, *components*, *vary_components*, *lo*, *hi*, *num*, *logspace=False*, *properties=None*, *progress=None*)

AFL.automation.prepare.factory.has_units

AFL.automation.prepare.factory.**has_units**(*value*)

AFL.automation.prepare.factory.is_concentration

AFL.automation.prepare.factory.**is_concentration**(*value*)

AFL.automation.prepare.factory.is_mass

AFL.automation.prepare.factory.**is_mass**(*value*)

AFL.automation.prepare.factory.is_molarity

AFL.automation.prepare.factory.**is_molarity**(*value*)

AFL.automation.prepare.factory.is_volume

AFL.automation.prepare.factory.**is_volume**(*value*)

AFL.automation.prepare.factory.listify

AFL.automation.prepare.factory.**listify**(*obj*)

Classes

Solution(name, components[, properties])	
product	product(*iterables, repeat=1) --> product object

AFL.automation.prepare.factory.Solution

class AFL.automation.prepare.factory.**Solution**(*name*, *components*, *properties=None*)

Bases: object

__init__(*name*, *components*, *properties=None*)

Methods

__init__(name, components[, properties])	
add_component_from_name(name[, properties, ...])	
contains(name)	
copy([name])	
from_dict(in_dict)	
measure_out(amount[, deplete])	Create solution with identical composition at new total mass/volume
rename_component(old_name, new_name[, inplace])	
set_mass(value)	Setter for inline mass changes
set_properties_from_dict([properties, inplace])	
set_volume(value)	Setter for inline volume changes
to_dict()	

Attributes

concentration	
mass	Total mass of mixture.
mass_fraction	Mass fraction of components in mixture
molarity	
size	
solutes	
solvent_density	
solvent_mass	
solvent_sld	
solvent_volume	
solvents	
volume	Total volume of mixture.
volume_fraction	Volume fraction of solvents in mixture

__init__(*name*, *components*, *properties=None*)

__hash__()

Needed so Solutions can be dictionary keys

to_dict()

classmethod from_dict(*in_dict*)

add_component_from_name(*name*, *properties=None*, *inplace=False*)

set_properties_from_dict(*properties=None*, *inplace=False*)

rename_component(*old_name*, *new_name*, *inplace=False*)

copy(*name=None*)

contains(*name*)

property size

property solutes

property solvents

__eq__(*other*)

'Compare the mass,volume, and composition of two mixtures

property mass

Total mass of mixture.

set_mass(*value*)

Setter for inline mass changes

property volume

Total volume of mixture. Only solvents are included in volume calculation

set_volume(*value*)

Setter for inline volume changes

property solvent_sld

property solvent_density

property solvent_volume

property solvent_mass

property mass_fraction

Mass fraction of components in mixture

Returns

- **mass_fraction** (*dict*)
- *Component mass fractions*

property volume_fraction

Volume fraction of solvents in mixture

Returns

- **solvent_fraction** (*dict*)
- *Component mass fractions*

property concentration

property molarity

measure_out(*amount*, *deplete=False*)

Create solution with identical composition at new total mass/volume

AFL.automation.prepare.factory.product

class AFL.automation.prepare.factory.**product**

Bases: object

product(*iterables, repeat=1) –> product object

Cartesian product of input iterables. Equivalent to nested for-loops.

For example, product(A, B) returns the same as: ((x,y) for x in A for y in B). The leftmost iterators are in the outermost for-loop, so the output tuples cycle in a manner similar to an odometer (with the rightmost element changing on every iteration).

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, product(A, repeat=4) means the same as product(A, A, A, A).

product('ab', range(3)) –> ('a',0) ('a',1) ('a',2) ('b',0) ('b',1) ('b',2) product((0,1), (0,1), (0,1)) –> (0,0,0) (0,0,1) (0,1,0) (0,1,1) (1,0,0) ...

__init__()

Methods

__init__()

AFL.automation.prepare.factory.**HD2OFactory**(*name*, *phi_D2O=None*, *sld=None*, *properties=None*)

Create a list of H2O/D2O solutions

AFL.automation.prepare.factory.**compositionSweepFactory**(*name*, *components*, *vary_components*, *lo*, *hi*, *num*, *logspace=False*, *properties=None*, *progress=None*)

AFL.automation.prepare.utilities

Functions

make_locs(slot, nrows, ncols)
make_wellplate_locs(slot, size)

AFL.automation.prepare.utilities.make_locs

AFL.automation.prepare.utilities.**make_locs**(*slot*, *nrows*, *ncols*)

AFL.automation.prepare.utilities.make_wellplate_locs

AFL.automation.prepare.utilities.**make_wellplate_locs**(*slot*, *size*)

AFL.automation.prepare.utilities.**make_locs**(*slot*, *nrows*, *ncols*)

AFL.automation.prepare.utilities.**make_wellplate_locs**(*slot*, *size*)

6.2.5 Sample

The Sample module provides functionality for experiment orchestration ‘Sample Servers’

AFL.automation.sample
AFL.automation.sample_env

AFL.automation.sample

Modules

CastingServer

AFL.automation.sample.CastingServer

Functions

ceil(x, /)	Return the ceiling of x as an Integral.
listify(obj)	
sqrt(x, /)	Return the square root of x.

AFL.automation.sample.CastingServer.ceil

AFL.automation.sample.CastingServer.**ceil**(*x*, /)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

AFL.automation.sample.CastingServer.listify

AFL.automation.sample.CastingServer.**listify**(*obj*)

AFL.automation.sample.CastingServer.sqrt

AFL.automation.sample.CastingServer.**sqrt**(*x*, /)

Return the square root of x.

Classes

CastingServer(prep_url[, overrides])	
Client([ip, port, username, interactive])	Communicate with APIServer
Driver(name[, defaults, overrides])	
OT2Client([ip, port, username, interactive])	Communicate with AFL-automation server on OT-2

AFL.automation.sample.CastingServer.CastingServer

class AFL.automation.sample.CastingServer.**CastingServer**(*prep_url*, *overrides=None*)

Bases: *Driver*

__init__(*prep_url*, *overrides=None*)

Methods

__init__(prep_url[, overrides])	
assign_targets(protocols, target_map)	
bulk_cast_films(**spec)	Spec should contain sample_name and a protocol
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
init_casting_manifest([attrs, overwrite])	
log_event(manifest_key, event[, write])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
prepare_and_cast(**sample)	
prepare_casting_stocks(**spec)	Spec should contain sample_name and a protocol
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	
update_status(value)	

Attributes

defaults

defaults = {'manifest': '/home/afl642/', 'manifest_cast': '/home/afl642/', 'manifest_prep': '/home/afl642/'}

__init__(*prep_url*, *overrides=None*)

init_casting_manifest(*attrs=None*, *overwrite=False*)

status()

update_status(*value*)

log_event(*manifest_key*, *event*, *write=True*)

assign_targets(*protocols*, *target_map*)

prepare_casting_stocks(***spec*)

Spec should contain sample_name and a protocol

bulk_cast_films(***spec*)

Spec should contain sample_name and a protocol

prepare_and_cast(***sample*)

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

AFL.automation.sample.CastingServer.Client

class AFL.automation.sample.CastingServer.**Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: object

Communicate with APIServer

This class provides an interface to generate HTTP REST requests that are sent to an APIServer, monitor the status of those requests, and retrieve the results of those requests. It is intended to be used as a client to the APIServer class.

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
clear_history()	
clear_queue()	

continues on next page

Table 58 – continued from previous page

debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
logged_in()	
login(username[, populate_commands])	
move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	

continues on next page

Table 58 – continued from previous page

set_config([interactive])
set_driver_object(**kw)
set_object([serialize])
unqueued_base(**kwargs)
wait([target_uuid, interval, for_history, ...])

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

classmethod from_server_name(*server_name*, ***kwargs*)

logged_in()

login(*username*, *populate_commands=True*)

driver_status()

get_queue()

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

get_quickbar()

server_cmd(*cmd*, ***kwargs*)

enqueued_base(***kwargs*)

unqueued_base(***kwargs*)

get_unqueued_commmands(*inherit_commands=True*)

get_queued_commmands(*inherit_commands=True*)

enqueue(*interactive=None*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_server_time()

query_driver(***kwargs*)

reset_queue_daemon()

pause(*state*)

clear_queue()

clear_history()

debug(*state*)

halt()

queue_state()

remove_item(*uuid*)

move_item(*uuid*, *pos*)

set_driver_object(***kw*)

get_driver_object(*name*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

set_object(*serialize=True*, ***kw*)

get_object(*name*, *serialize=True*)

AFL.automation.sample.CastingServer.Driver

class AFL.automation.sample.CastingServer.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.sample.CastingServer.OT2Client

class AFL.automation.sample.CastingServer.**OT2Client**(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Bases: *Client*

Communicate with AFL-automation server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the AFL-automation server

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

Methods

__init__([ip, port, username, interactive])	
aspirate_rate(rate)	
clear_history()	
clear_queue()	
debug(state)	
deposit_obj(obj[, uid])	Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated
dispense_rate(rate)	
driver_status()	
enqueue([interactive])	
enqueued_base(**kwargs)	
from_server_name(server_name, **kwargs)	
get_config(name[, print_console, interactive])	
get_driver_object(name)	
get_object(name[, serialize])	
get_queue()	
get_queued_commmands([inherit_commands])	
get_quickbar()	
get_server_time()	
get_unqueued_commmands([inherit_commands])	
halt()	
home()	
load_instrument(name, mount, tip_rack_slots)	
load_labware(name, slot)	
logged_in()	
login(username[, populate_commands])	

continues on next page

Table 59 – continued from previous page

move_item(uuid, pos)	
pause(state)	
query_driver(**kwargs)	
queue_state()	
remove_item(uuid)	
reset_queue_daemon()	
reset_tipracks([mount])	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving
server_cmd(cmd, **kwargs)	
set_config([interactive])	
set_driver_object(**kw)	
set_object([serialize])	
transfer(source, dest, volume[, interactive])	Transfer fluid from one location to another
unqueued_base(**kwargs)	
wait([target_uuid, interval, for_history, ...])	

transfer(*source*, *dest*, *volume*, *interactive=None*, ***kwargs*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str Source wells to transfer from. Wells should be specified as three*) – character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

reset_tipracks(*mount='both'*)

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

aspirate_rate(*rate*)

dispense_rate(*rate*)

home()

__init__(*ip=None*, *port='5000'*, *username=None*, *interactive=False*)

clear_history()

clear_queue()

debug(*state*)

deposit_obj(*obj*, *uid=None*)

Deposit an object in the dropbox obj : object, the object to deposit id : str, the uuid to deposit the object under if not specified, a new uuid will be generated

driver_status()

enqueue(*interactive=None*, ***kwargs*)

enqueued_base(***kwargs*)

classmethod from_server_name(*server_name*, ***kwargs*)

get_config(*name*, *print_console=True*, *interactive=None*)

get_driver_object(*name*)

get_object(*name*, *serialize=True*)

get_queue()

get_queued_commmands(*inherit_commands=True*)

get_quickbar()

get_server_time()

get_unqueued_commmands(*inherit_commands=True*)

halt()

logged_in()

login(*username*, *populate_commands=True*)

move_item(*uuid*, *pos*)

pause(*state*)

query_driver(***kwargs*)

queue_state()

remove_item(*uuid*)

reset_queue_daemon()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox id : str, the uuid of the object to retrieve delete : bool, if True, delete the object after retrieving

server_cmd(*cmd*, ***kwargs*)

set_config(*interactive=None*, ***kwargs*)

set_driver_object(***kw*)

set_object(*serialize=True*, ***kw*)

unqueued_base(***kwargs*)

wait(*target_uuid=None*, *interval=0.1*, *for_history=True*, *first_check_delay=5.0*)

class AFL.automation.sample.CastingServer.**CastingServer**(*prep_url*, *overrides=None*)

defaults = {'manifest': '/home/afl642/', 'manifest_cast': '/home/afl642/', 'manifest_prep': '/home/afl642/'}

__init__(*prep_url*, *overrides=None*)

init_casting_manifest(*attrs=None*, *overwrite=False*)

status()

update_status(*value*)

log_event(*manifest_key*, *event*, *write=True*)

assign_targets(*protocols*, *target_map*)

prepare_casting_stocks(***spec*)

Spec should contain sample_name and a protocol

bulk_cast_films(***spec*)

Spec should contain sample_name and a protocol

prepare_and_cast(***sample*)

AFL.automation.sample_env

Modules

TemperatureDeck

AFL.automation.sample_env.TemperatureDeck

Classes

Driver(name[, defaults, overrides])
TemperatureDeck([overrides])

AFL.automation.sample_env.TemperatureDeck.Driver

class AFL.automation.sample_env.TemperatureDeck.**Driver**(*name*, *defaults=None*, *overrides=None*)

Bases: object

__init__(*name*, *defaults=None*, *overrides=None*)

Methods

__init__(name[, defaults, overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
status()	
unqueued()	

unqueued()

queued()

quickbar()

__init__(*name*, *defaults=None*, *overrides=None*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

set_config(***kwargs*)

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

get_sample()

reset_sample()

status()

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

execute(***kwargs*)

set_object(*serialized=True*, ***kw*)

get_object(*name*, *serialize=True*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

AFL.automation.sample_env.TemperatureDeck.TemperatureDeck

class AFL.automation.sample_env.TemperatureDeck.**TemperatureDeck**(*overrides=None*)

Bases: *Driver*

__init__(*overrides=None*)

Methods

__init__([overrides])	
deposit_obj(obj[, uid])	Store an object in the dropbox
execute(**kwargs)	
gather_defaults()	Gather all inherited static class-level dictionaries called default.
get_config(name[, print_console])	
get_configs([print_console])	
get_object(name[, serialize])	
get_sample()	
move_temp(temperature[, wait, tolerance])	
post_execute(**kwargs)	Executed after each call to execute
pre_execute(**kwargs)	Executed before each call to execute
queued()	
quickbar()	
read_temp()	
reset_sample()	
retrieve_obj(uid[, delete])	Retrieve an object from the dropbox
set_config(**kwargs)	
set_data(data)	Set data in the DataPacket object
set_object([serialized])	
set_sample(sample_name[, sample_uuid])	
set_temp(sp)	
status()	
unqueued()	

Attributes

defaults

defaults = {'serial_port': '/dev/ttyACM0', 'temperature_move_sleep': 0.2, 'temperature_move_timeout': 900}

__init__(*overrides=None*)

set_temp(*sp*)

move_temp(*temperature*, *wait=30*, *tolerance=0.1*)

status()

read_temp()

deposit_obj(*obj*, *uid=None*)

Store an object in the dropbox

Parameters

- **obj** (*object*) – The object to store in the dropbox
- **uid** (*str*) – The uuid to store the object under

execute(***kwargs*)

classmethod gather_defaults()

Gather all inherited static class-level dictionaries called default.

get_config(*name*, *print_console=False*)

get_configs(*print_console=False*)

get_object(*name*, *serialize=True*)

get_sample()

post_execute(***kwargs*)

Executed after each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

pre_execute(***kwargs*)

Executed before each call to execute

All of the kwargs passed to execute are also pass to this method. It is expected that this method be overridden by subclasses.

queued()

quickbar()

reset_sample()

retrieve_obj(*uid*, *delete=True*)

Retrieve an object from the dropbox

Parameters
uid (*str*) – The uuid of the file to retrieve

set_config(***kwargs*)

set_data(*data: dict*)

Set data in the DataPacket object

Parameters

- **data** (*dict*) – Dictionary of data to store in the driver object
- **variables** (*Note! if the keys in data are not system or sample*)

:param : :param they will be erased at the end of this function call.:

set_object(*serialized=True*, ***kw*)

set_sample(*sample_name*, *sample_uuid=None*, ***kwargs*)

unqueued()

class AFL.automation.sample_env.TemperatureDeck.**TemperatureDeck**(*overrides=None*)

defaults = {'serial_port': '/dev/ttyACM0', 'temperature_move_sleep': 0.2, 'temperature_move_timeout': 900}

__init__(*overrides=None*)

set_temp(*sp*)

move_temp(*temperature*, *wait=30*, *tolerance=0.1*)

status()

read_temp()

6.2.6 Shared

The Shared module provides common utilities and functionality used across the AFL-automation framework.

AFL.automation.shared

AFL.automation.shared

Modules

DataLabelerWidget	
DatasetWidget	
DiffractionLabeler	
MutableQueue	
PersistentConfig	
ServerDiscovery	
exceptions	
serialization	
units	
utilities	
widgetui	

AFL.automation.shared.DataLabelerWidget

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.shared.DataLabelerWidget.sqrt

AFL.automation.shared.DataLabelerWidget.**sqrt**(*x*, /)

Return the square root of x.

Classes

DataLabelerModel(dataset)	
DataLabelerView()	
DataLabelerWidget(input_dataset, ...[, ...])	
OrdinalEncoder(*[, categories, dtype, ...])	Encode categorical features as an integer array.

AFL.automation.shared.DataLabelerWidget.DataLabelerModel

class AFL.automation.shared.DataLabelerWidget.**DataLabelerModel**(*dataset: Dataset*)

Bases: object

__init__(*dataset: Dataset*)

Methods

__init__(dataset)
get_peaks(model[, qstar, max_order])
init_models()
ordinal_phase_labels()

__init__(*dataset: Dataset*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

AFL.automation.shared.DataLabelerWidget.DataLabelerView

class AFL.automation.shared.DataLabelerWidget.**DataLabelerView**

Bases: object

__init__()

Methods

__init__()
add_vertical_line(x[, y0, y1, row, col, line_kw])
remove_vertical_lines()
run(x, y, all_compositions, composition, ...)
update_composition_colors(colors)
update_plot(x, y, composition)

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_composition_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *ternary*, *components*)

AFL.automation.shared.DataLabelerWidget.DataLabelerWidget

class AFL.automation.shared.DataLabelerWidget.**DataLabelerWidget**(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

Bases: object

__init__(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

Parameters

- **dataset** (*xr.Dataset*) – Dataset from AFL
- **sas_variable** (*str*) – Name of data variable in the *xarray.Dataset* that holds the scattering data
- **composition_variable** (*str | List[str]*) – Name of data variable in the *xarray.Dataset* that holds the composition. If the composition is split across multiple variables, pass in a list of variables.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to each sample or measurement. This is typically named 'sample' in much of the AFL agent codebase.
- **fit_variable** (*Optional[str]*) – If not none, this data will be plotted along with the sas_variable data. This data variable should have the same shape as sas_variable.

Methods

__init__(input_dataset, sas_variable, ...[, ...])	
change_model_callback(data)	
change_norder_callback(data)	
change_qstar_callback(figure, location, click)	
composition_click_callback(figure, location, ...)	
draw_peaks(peaks)	
goto_callback(click)	
label(label)	
next_button_callback(click)	
prev_button_callback(click)	
run()	
update_plot()	

__init__(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

Parameters

- **dataset** (*xr.Dataset*) – Dataset from AFL
- **sas_variable** (*str*) – Name of data variable in the *xarray.Dataset* that holds the scattering data
- **composition_variable** (*str | List[str]*) – Name of data variable in the *xarray.Dataset* that holds the composition. If the composition is split across multiple variables, pass in a list of variables.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to each sample or measurement. This is typically named 'sample' in much of the AFL agent codebase.
- **fit_variable** (*Optional[str]*) – If not none, this data will be plotted along with the sas_variable data. This data variable should have the same shape as sas_variable.

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

composition_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

AFL.automation.shared.DataLabelerWidget.OrdinalEncoder

class AFL.automation.shared.DataLabelerWidget.**OrdinalEncoder**(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

Bases: OneToOneFeatureMixin, _BaseEncoder

Encode categorical features as an integer array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are converted to ordinal integers. This results in a single column of integers (0 to n_categories - 1) per feature.

Read more in the User Guide. For a comparison of different encoders, refer to: sphx_glr_auto_examples_preprocessing_plot_target_encoder.py.

Added in version 0.20.

Parameters

- **categories** (*'auto' or a list of array-like, default='auto'*) – Categories (unique values) per feature:
  - 'auto' : Determine categories automatically from the training data.
  - list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values, and should be sorted in case of numeric values.

  The used categories can be found in the `categories_` attribute.
- **dtype** (*number type, default=np.float64*) – Desired dtype of output.
- **handle_unknown** (*{'error', 'use_encoded_value'}, default='error'*) – When set to 'error' an error will be raised in case an unknown categorical feature is present during transform. When set to 'use_encoded_value', the encoded value of unknown categories will be set to the value given for the parameter *unknown_value*. In *inverse_transform()*, an unknown category will be denoted as None.

  Added in version 0.24.
- **unknown_value** (*int or np.nan, default=None*) – When the parameter handle_unknown is set to 'use_encoded_value', this parameter is required and will set the encoded value of unknown categories. It has to be distinct from the values used to encode any of the categories in *fit*. If set to np.nan, the *dtype* parameter must be a float dtype.

Added in version 0.24.

- **encoded_missing_value** (*int or np.nan, default=np.nan*) – Encoded value of missing categories. If set to *np.nan*, then the *dtype* parameter must be a float dtype.

  Added in version 1.1.

- **min_frequency** (*int or float, default=None*) – Specifies the minimum frequency below which a category will be considered infrequent.

  – If *int*, categories with a smaller cardinality will be considered infrequent.

  – If *float*, categories with a smaller cardinality than *min_frequency* * *n_samples* will be considered infrequent.

  Added in version 1.3: Read more in the User Guide.

- **max_categories** (*int, default=None*) – Specifies an upper limit to the number of output categories for each input feature when considering infrequent categories. If there are infrequent categories, *max_categories* includes the category representing the infrequent categories along with the frequent categories. If *None*, there is no limit to the number of output features.

  max_categories do **not** take into account missing or unknown categories. Setting *unknown_value* or *encoded_missing_value* to an integer will increase the number of unique integer codes by one each. This can result in up to *max_categories* + *2* integer codes.

  Added in version 1.3: Read more in the User Guide.

categories_

The categories of each feature determined during `fit` (in order of the features in X and corresponding with the output of `transform`). This does not include categories that weren't seen during `fit`.

Type
list of arrays

n_features_in_

Number of features seen during fit.

Added in version 1.0.

Type
int

feature_names_in_

Names of features seen during fit. Defined only when *X* has feature names that are all strings.

Added in version 1.0.

Type
ndarray of shape (*n_features_in_*,)

infrequent_categories_

Defined only if infrequent categories are enabled by setting *min_frequency* or *max_categories* to a non-default value. *infrequent_categories_[i]* are the infrequent categories for feature *i*. If the feature *i* has no infrequent categories *infrequent_categories_[i]* is None.

Added in version 1.3.

Type
list of ndarray

See also

OneHotEncoder
Performs a one-hot encoding of categorical features. This encoding is suitable for low to medium cardinality categorical variables, both in supervised and unsupervised settings.

TargetEncoder
Encodes categorical features using supervised signal in a classification or regression pipeline. This encoding is typically suitable for high cardinality categorical variables.

LabelEncoder
Encodes target labels with values between 0 and `n_classes-1`.

Notes

With a high proportion of *nan* values, inferring categories becomes slow with Python versions before 3.10. The handling of *nan* values was improved from Python 3.10 onwards, (c.f. bpo-43475).

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to an ordinal encoding.

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> enc = OrdinalEncoder()
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
OrdinalEncoder()
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 3], ['Male', 1]])
array([[0., 2.],
       [1., 0.]])
```

```
>>> enc.inverse_transform([[1, 0], [0, 1]])
array([['Male', 1],
       ['Female', 2]], dtype=object)
```

By default, *OrdinalEncoder* is lenient towards missing values by propagating them.

```
>>> import numpy as np
>>> X = [['Male', 1], ['Female', 3], ['Female', np.nan]]
>>> enc.fit_transform(X)
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0., nan]])
```

You can use the parameter *encoded_missing_value* to encode missing values.

```
>>> enc.set_params(encoded_missing_value=-1).fit_transform(X)
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0., -1.]])
```

Infrequent categories are enabled by setting *max_categories* or *min_frequency*. In the following example, “a” and “d” are considered infrequent and grouped together into a single category, “b” and “c” are their own categories, unknown values are encoded as 3 and missing values are encoded as 4.

```
>>> X_train = np.array(
...     [["a"] * 5 + ["b"] * 20 + ["c"] * 10 + ["d"] * 3 + [np.nan]],
...     dtype=object).T
>>> enc = OrdinalEncoder(
...     handle_unknown="use_encoded_value", unknown_value=3,
...     max_categories=3, encoded_missing_value=4)
>>> _ = enc.fit(X_train)
>>> X_test = np.array([["a"], ["b"], ["c"], ["d"], ["e"], [np.nan]], dtype=object)
>>> enc.transform(X_test)
array([[2.],
       [0.],
       [1.],
       [2.],
       [3.],
       [4.]])
```

__init__(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

Methods

__init__(*[, categories, dtype, ...])	
fit(X[, y])	Fit the OrdinalEncoder to X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Get output feature names for transformation.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Convert the data back to the original representation.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to ordinal codes.

Attributes

*infrequent_categories_*	Infrequent categories for each feature.

__init__(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

fit(*X*, *y=None*)

Fit the OrdinalEncoder to X.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – The data to determine the categories of each feature.

- **y** (*None*) – Ignored. This parameter exists only for compatibility with `Pipeline`.

Returns
self – Fitted encoder.

Return type
object

`transform`(*X*)

Transform X to ordinal codes.

Parameters
X (*array-like of shape (n_samples, n_features)*) – The data to encode.

Returns
X_out – Transformed input.

Return type
ndarray of shape (n_samples, n_features)

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters
X (*array-like of shape (n_samples, n_encoded_features)*) – The transformed data.

Returns
X_tr – Inverse transformed array.

Return type
ndarray of shape (n_samples, n_features)

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns
X_new – Transformed array.

Return type
ndarray array of shape (n_samples, n_features_new)

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation.

Parameters
input_features (*array-like of str or None, default=None*) – Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then the following input feature names are generated: *["x0", "x1", ..., "x(n_features_in_ - 1)"]*.

- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns
feature_names_out – Same as input features.

Return type
ndarray of str objects

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params(deep=True)`

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`property infrequent_categories_`

Infrequent categories for each feature.

`set_output(*, transform=None)`

Set output container.

See sphx_glr_auto_examples_miscellaneous_plot_set_output.py for an example on how to use the API.

Parameters
transform (*{"default", "pandas", "polars"}, default=None*) – Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns
self – Estimator instance.

Return type
estimator instance

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

class AFL.automation.shared.DataLabelerWidget.**DataLabelerWidget**(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

__init__(*input_dataset: Dataset*, *sas_variable: str*, *composition_variable: str | List[str]*, *sample_dim: str = 'sample'*, *fit_variable: str | None = None*)

Parameters

- **dataset** (*xr.Dataset*) – Dataset from AFL
- **sas_variable** (*str*) – Name of data variable in the *xarray.Dataset* that holds the scattering data
- **composition_variable** (*str | List[str]*) – Name of data variable in the *xarray.Dataset* that holds the composition. If the composition is split across multiple variables, pass in a list of variables.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to each sample or measurement. This is typically named 'sample' in much of the AFL agent codebase.
- **fit_variable** (*Optional[str]*) – If not none, this data will be plotted along with the sas_variable data. This data variable should have the same shape as sas_variable.

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

composition_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

class AFL.automation.shared.DataLabelerWidget.**DataLabelerModel**(*dataset: Dataset*)

__init__(*dataset: Dataset*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

class AFL.automation.shared.DataLabelerWidget.**DataLabelerView**

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_composition_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *ternary*, *components*)

AFL.automation.shared.DatasetWidget

Classes

DatasetWidget(dataset[, sample_dim, ...])	
DatasetWidget_Model(dataset, sample_dim)	
DatasetWidget_View([...])	
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

AFL.automation.shared.DatasetWidget.DatasetWidget

class AFL.automation.shared.DatasetWidget.**DatasetWidget**(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

Bases: object

__init__(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

Interactive widget for viewing compositionally varying scattering data

Parameters

- **dataset** (*xr.Dataset*) – *xarray.Dataset* containing scattering data and compositions to be plotted.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to sample variation, typically “sample”
- **comps_variable** (*Optional[str]*) – The name of the *xarray* variable to plot as compositional data. Optional, if not specified, can be customized in the GUI.

  Only the first two columns of the data will be used in the plot. If the compositions are in separate `xarray.DataArray`s, they should be grouped into single `xarray.DataArray`s like so:

  `python ds['comps'] = ds[['A','B','C']].to_array('component'). transpose(...,'component') `
- **comps_color_variable** (*Optional[str]*) – The name of the *xarray* variable to use as the colorscale of the compositional data scatter plot. Optional, if not specified, can be customized in the GUI.
- **xmin** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **xmax** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **Usage**
- **-----**
- **```python** –
- **DatasetWidget(ds)** (*widget =*)
- **widget.run()**
- ``` _

Methods

__init__(dataset[, sample_dim, ...])	Interactive widget for viewing compositionally varying scattering data
apply_isel(*args)	
apply_sel(*args)	
combine_vars(*args)	
composition_click_callback(figure, location, ...)	
extract_var(*args)	
get_comps()	
goto_callback(*args)	
initialize_plots(*args)	
next_button_callback(*args)	
prev_button_callback(*args)	
reset_dataset(*args)	
run()	
update_colors(*args)	
update_composition_plot()	
update_dropdowns(*args)	
update_extract_coords(change)	
update_plots()	
update_sample_dim(*args)	
update_scattering_plot()	

__init__(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

Interactive widget for viewing compositionally varying scattering data

Parameters

- **dataset** (*xr.Dataset*) – *xarray.Dataset* containing scattering data and compositions to be plotted.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to sample variation,

typically “sample”

- **comps_variable** (*Optional[str]*) – The name of the *xarray* variable to plot as compositional data. Optional, if not specified, can be customized in the GUI.

  Only the first two columns of the data will be used in the plot. If the compositions are in separate `xarray.DataArray`s, they should be grouped into single `xarray.DataArray`s like so:

  `python ds['comps'] = ds[['A','B','C']].to_array('component'). transpose(...,'component') `

- **comps_color_variable** (*Optional[str]*) – The name of the *xarray* variable to use as the colorscale of the compositional data scatter plot. Optional, if not specified, can be customized in the GUI.
- **xmin** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **xmax** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **Usage**
- **-----**
- **```python** –
- **DatasetWidget(ds)** (*widget =*)
- **widget.run()**
- **``` _**

next_button_callback(**args*)

prev_button_callback(**args*)

goto_callback(**args*)

composition_click_callback(*figure*, *location*, *click*)

update_composition_plot()

update_scattering_plot()

update_plots()

get_comps()

initialize_plots(**args*)

update_colors(**args*)

apply_sel(**args*)

apply_isel(**args*)

extract_var(**args*)

combine_vars(**args*)

reset_dataset(**args*)

update_dropdowns(**args*)

update_sample_dim(**args*)

update_extract_coords(*change*)

run()

AFL.automation.shared.DatasetWidget.DatasetWidget_Model

class AFL.automation.shared.DatasetWidget.**DatasetWidget_Model**(*dataset: Dataset*, *sample_dim: str*)

Bases: object

__init__(*dataset: Dataset*, *sample_dim: str*)

Methods

__init__(dataset, sample_dim)	
apply_isel(kw)	
apply_sel(kw)	
combine_vars(combined_var, to_combine_vars)	
extract_var(extract_from_var, extract_from_coord)	ex-
get_composition(variable)	
get_non_sample_dims(var)	
get_scattering(variable, index)	
reset_dataset()	
split_vars()	Heuristically try to split vars into categories

Attributes

dataset

__init__(*dataset: Dataset*, *sample_dim: str*)

property dataset

reset_dataset()

split_vars()

Heuristically try to split vars into categories

get_non_sample_dims(*var: str*)

apply_sel(*kw*)

apply_isel(*kw*)

combine_vars(*combined_var: str*, *to_combine_vars: List[str]*)

extract_var(*extract_from_var: str*, *extract_from_coord: str*)

get_composition(*variable*)

get_scattering(*variable*, *index*)

AFL.automation.shared.DatasetWidget.DatasetWidget_View

class AFL.automation.shared.DatasetWidget.**DatasetWidget_View**(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

Bases: object

__init__(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

Methods

__init__([initial_scatt_variables, ...])
init_buttons()
init_checkboxes()
init_dropdowns()
init_inputs()
init_plots()
plot_comp(x, y[, z, xname, yname, zname, colors])
plot_sas(x, y[, name, append])
run()
update_colorscale([colors])
update_dropdowns([sample_vars, scatt_vars, ...])
update_selected(**kw)

__init__(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

update_colorscale(*colors=None*)

update_selected(***kw*)

update_dropdowns(*sample_vars=None*, *scatt_vars=None*, *comp_vars=None*)

plot_sas(*x*, *y*, *name='SAS'*, *append=False*)

plot_comp(*x*, *y*, *z=None*, *xname='x'*, *yname='y'*, *zname='z'*, *colors=None*)

init_plots()

init_buttons()

init_checkboxes()

init_dropdowns()

init_inputs()

run()

AFL.automation.shared.DatasetWidget.defaultdict

class AFL.automation.shared.DatasetWidget.**defaultdict**

Bases: dict

defaultdict(default_factory=None, /, [...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D.

default_factory

Factory for default value called by __missing__().

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class AFL.automation.shared.DatasetWidget.**DatasetWidget**(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

__init__(*dataset: Dataset*, *sample_dim: str = 'sample'*, *scatt_variables: List[str] | None = None*, *comps_variable: str | None = None*, *comps_color_variable: str | None = None*, *xmin: float = 0.001*, *xmax: float = 1.0*)

Interactive widget for viewing compositionally varying scattering data

Parameters

- **dataset** (*xr.Dataset*) – *xarray.Dataset* containing scattering data and compositions to be plotted.
- **sample_dim** (*str*) – The name of the *xarray* dimension corresponding to sample variation, typically "sample"
- **comps_variable** (*Optional[str]*) – The name of the *xarray* variable to plot as compositional data. Optional, if not specified, can be customized in the GUI.

  Only the first two columns of the data will be used in the plot. If the compositions are in separate `xarray.DataArray`s, they should be grouped into single `xarray.DataArray`s like so:

  ```
  `python ds['comps'] = ds[['A','B','C']].to_array('component').
  transpose(...,'component') `
  ```
- **comps_color_variable** (*Optional[str]*) – The name of the *xarray* variable to use as the colorscale of the compositional data scatter plot. Optional, if not specified, can be customized in the GUI.
- **xmin** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **xmax** (*float*) – Set the default q-range of the scattering data. Can be customized in the GUI
- **Usage**
- **-----**
- **```python** –
- **DatasetWidget(ds)** (*widget =*)

- **widget.run()**
- ``` _

next_button_callback(**args*)

prev_button_callback(**args*)

goto_callback(**args*)

composition_click_callback(*figure*, *location*, *click*)

update_composition_plot()

update_scattering_plot()

update_plots()

get_comps()

initialize_plots(**args*)

update_colors(**args*)

apply_sel(**args*)

apply_isel(**args*)

extract_var(**args*)

combine_vars(**args*)

reset_dataset(**args*)

update_dropdowns(**args*)

update_sample_dim(**args*)

update_extract_coords(*change*)

run()

class AFL.automation.shared.DatasetWidget.**DatasetWidget_Model**(*dataset: Dataset*, *sample_dim: str*)

__init__(*dataset: Dataset*, *sample_dim: str*)

property dataset

reset_dataset()

split_vars()

Heuristically try to split vars into categories

get_non_sample_dims(*var: str*)

apply_sel(*kw*)

apply_isel(*kw*)

combine_vars(*combined_var: str*, *to_combine_vars: List[str]*)

extract_var(*extract_from_var: str*, *extract_from_coord: str*)

get_composition(*variable*)

get_scattering(*variable*, *index*)

class AFL.automation.shared.DatasetWidget.**DatasetWidget_View**(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

__init__(*initial_scatt_variables: List[str] | None = None*, *initial_comps_variable: str | None = None*, *initial_comps_color_variable: str | None = None*)

update_colorscale(*colors=None*)

update_selected(***kw*)

update_dropdowns(*sample_vars=None*, *scatt_vars=None*, *comp_vars=None*)

plot_sas(*x*, *y*, *name='SAS'*, *append=False*)

plot_comp(*x*, *y*, *z=None*, *xname='x'*, *yname='y'*, *zname='z'*, *colors=None*)

init_plots()

init_buttons()

init_checkboxes()

init_dropdowns()

init_inputs()

run()

AFL.automation.shared.DiffractionLabeler

Functions

sqrt(x, /)	Return the square root of x.

AFL.automation.shared.DiffractionLabeler.sqrt

AFL.automation.shared.DiffractionLabeler.**sqrt**(*x*, */*)

Return the square root of x.

Classes

DiffractionLabeler(saxs_data, ...)	
DiffractionLabelerModel(saxs_data, ...)	
DiffractionLabelerView()	
OrdinalEncoder(*[, categories, dtype, ...])	Encode categorical features as an integer array.

AFL.automation.shared.DiffractionLabeler.DiffractionLabeler

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabeler**(*saxs_data*, *composition_data*, *possible_phase_labels*)

Bases: object

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

Methods

__init__(saxs_data, composition_data, ...)
change_model_callback(data)
change_norder_callback(data)
change_qstar_callback(figure, location, click)
draw_peaks(peaks)
goto_callback(click)
label(label)
next_button_callback(click)
prev_button_callback(click)
run()
ternary_click_callback(figure, location, click)
update_plot()

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

ternary_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

AFL.automation.shared.DiffractionLabeler.DiffractionLabelerModel

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerModel**(*saxs_data*, *composition_data*, *possible_phase_labels*)

Bases: object

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

Methods

__init__(saxs_data, composition_data, ...)
get_peaks(model[, qstar, max_order])
init_models()
ordinal_phase_labels()

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

AFL.automation.shared.DiffractionLabeler.DiffractionLabelerView

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerView**

Bases: object

__init__()

Methods

__init__()
add_vertical_line(x[, y0, y1, row, col, line_kw])
remove_vertical_lines()
run(x, y, all_compositions, composition, ...)
update_plot(x, y, composition)
update_ternary_colors(colors)

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_ternary_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *possible_phase_labels*)

AFL.automation.shared.DiffractionLabeler.OrdinalEncoder

class AFL.automation.shared.DiffractionLabeler.**OrdinalEncoder**(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

Bases: OneToOneFeatureMixin, _BaseEncoder

Encode categorical features as an integer array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are converted to ordinal integers. This results in a single column of integers (0 to n_categories - 1) per feature.

Read more in the User Guide. For a comparison of different encoders, refer to: sphx_glr_auto_examples_preprocessing_plot_target_encoder.py.

Added in version 0.20.

Parameters

- **categories** (*'auto' or a list of array-like, default='auto'*) – Categories (unique values) per feature:
  - 'auto' : Determine categories automatically from the training data.
  - list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values, and should be sorted in case of numeric values.

  The used categories can be found in the `categories_` attribute.
- **dtype** (*number type, default=np.float64*) – Desired dtype of output.
- **handle_unknown** (*{'error', 'use_encoded_value'}, default='error'*) – When set to 'error' an error will be raised in case an unknown categorical feature is present during transform. When set to 'use_encoded_value', the encoded value of unknown categories will be set to the value given for the parameter *unknown_value*. In *inverse_transform()*, an unknown category will be denoted as None.

  Added in version 0.24.
- **unknown_value** (*int or np.nan, default=None*) – When the parameter handle_unknown is set to 'use_encoded_value', this parameter is required and will set the encoded value of unknown categories. It has to be distinct from the values used to encode any of the categories in *fit*. If set to np.nan, the *dtype* parameter must be a float dtype.

Added in version 0.24.

- **encoded_missing_value** (*int or np.nan, default=np.nan*) – Encoded value of missing categories. If set to *np.nan*, then the *dtype* parameter must be a float dtype.

  Added in version 1.1.

- **min_frequency** (*int or float, default=None*) – Specifies the minimum frequency below which a category will be considered infrequent.

  – If *int*, categories with a smaller cardinality will be considered infrequent.

  – If *float*, categories with a smaller cardinality than *min_frequency* * *n_samples* will be considered infrequent.

  Added in version 1.3: Read more in the User Guide.

- **max_categories** (*int, default=None*) – Specifies an upper limit to the number of output categories for each input feature when considering infrequent categories. If there are infrequent categories, *max_categories* includes the category representing the infrequent categories along with the frequent categories. If *None*, there is no limit to the number of output features.

  max_categories do **not** take into account missing or unknown categories. Setting *unknown_value* or *encoded_missing_value* to an integer will increase the number of unique integer codes by one each. This can result in up to *max_categories* + *2* integer codes.

  Added in version 1.3: Read more in the User Guide.

categories_

The categories of each feature determined during `fit` (in order of the features in X and corresponding with the output of `transform`). This does not include categories that weren't seen during `fit`.

Type
list of arrays

n_features_in_

Number of features seen during fit.

Added in version 1.0.

Type
int

feature_names_in_

Names of features seen during fit. Defined only when *X* has feature names that are all strings.

Added in version 1.0.

Type
ndarray of shape (*n_features_in_*,)

infrequent_categories_

Defined only if infrequent categories are enabled by setting *min_frequency* or *max_categories* to a non-default value. *infrequent_categories_[i]* are the infrequent categories for feature *i*. If the feature *i* has no infrequent categories *infrequent_categories_[i]* is None.

Added in version 1.3.

Type
list of ndarray

See also

OneHotEncoder
: Performs a one-hot encoding of categorical features. This encoding is suitable for low to medium cardinality categorical variables, both in supervised and unsupervised settings.

TargetEncoder
: Encodes categorical features using supervised signal in a classification or regression pipeline. This encoding is typically suitable for high cardinality categorical variables.

LabelEncoder
: Encodes target labels with values between 0 and `n_classes-1`.

Notes

With a high proportion of *nan* values, inferring categories becomes slow with Python versions before 3.10. The handling of *nan* values was improved from Python 3.10 onwards, (c.f. bpo-43475).

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to an ordinal encoding.

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> enc = OrdinalEncoder()
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
OrdinalEncoder()
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 3], ['Male', 1]])
array([[0., 2.],
       [1., 0.]])
```

```
>>> enc.inverse_transform([[1, 0], [0, 1]])
array([['Male', 1],
       ['Female', 2]], dtype=object)
```

By default, *OrdinalEncoder* is lenient towards missing values by propagating them.

```
>>> import numpy as np
>>> X = [['Male', 1], ['Female', 3], ['Female', np.nan]]
>>> enc.fit_transform(X)
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0., nan]])
```

You can use the parameter *encoded_missing_value* to encode missing values.

```
>>> enc.set_params(encoded_missing_value=-1).fit_transform(X)
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0., -1.]])
```

Infrequent categories are enabled by setting *max_categories* or *min_frequency*. In the following example, “a” and “d” are considered infrequent and grouped together into a single category, “b” and “c” are their own categories, unknown values are encoded as 3 and missing values are encoded as 4.

```
>>> X_train = np.array(
...     [["a"] * 5 + ["b"] * 20 + ["c"] * 10 + ["d"] * 3 + [np.nan]],
...     dtype=object).T
>>> enc = OrdinalEncoder(
...     handle_unknown="use_encoded_value", unknown_value=3,
...     max_categories=3, encoded_missing_value=4)
>>> _ = enc.fit(X_train)
>>> X_test = np.array([["a"], ["b"], ["c"], ["d"], ["e"], [np.nan]], dtype=object)
>>> enc.transform(X_test)
array([[2.],
       [0.],
       [1.],
       [2.],
       [3.],
       [4.]])
```

__init__(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

Methods

__init__(*[, categories, dtype, ...])	
fit(X[, y])	Fit the OrdinalEncoder to X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Get output feature names for transformation.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Convert the data back to the original representation.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to ordinal codes.

Attributes

*infrequent_categories_*	Infrequent categories for each feature.

__init__(*, *categories='auto'*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=nan*, *min_frequency=None*, *max_categories=None*)

fit(*X*, *y=None*)

Fit the OrdinalEncoder to X.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – The data to determine the categories of each feature.

- **y** (*None*) – Ignored. This parameter exists only for compatibility with `Pipeline`.

Returns
self – Fitted encoder.

Return type
object

`transform`(*X*)

Transform X to ordinal codes.

Parameters
X (*array-like of shape (n_samples, n_features)*) – The data to encode.

Returns
X_out – Transformed input.

Return type
ndarray of shape (n_samples, n_features)

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters
X (*array-like of shape (n_samples, n_encoded_features)*) – The transformed data.

Returns
X_tr – Inverse transformed array.

Return type
ndarray of shape (n_samples, n_features)

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns
X_new – Transformed array.

Return type
ndarray array of shape (n_samples, n_features_new)

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation.

Parameters
input_features (*array-like of str or None, default=None*) – Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then the following input feature names are generated: *["x0", "x1", ..., "x(n_features_in_ - 1)"]*.

- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns
feature_names_out – Same as input features.

Return type
ndarray of str objects

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params(deep=True)`

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`property infrequent_categories_`

Infrequent categories for each feature.

`set_output(*, transform=None)`

Set output container.

See sphx_glr_auto_examples_miscellaneous_plot_set_output.py for an example on how to use the API.

Parameters
transform (*{"default", "pandas", "polars"}, default=None*) – Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns
self – Estimator instance.

Return type
estimator instance

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabeler**(*saxs_data*, *composition_data*, *possible_phase_labels*)

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

next_button_callback(*click*)

prev_button_callback(*click*)

goto_callback(*click*)

ternary_click_callback(*figure*, *location*, *click*)

update_plot()

draw_peaks(*peaks*)

change_qstar_callback(*figure*, *location*, *click*)

change_model_callback(*data*)

change_norder_callback(*data*)

label(*label*)

run()

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerModel**(*saxs_data*, *composition_data*, *possible_phase_labels*)

__init__(*saxs_data*, *composition_data*, *possible_phase_labels*)

ordinal_phase_labels()

init_models()

get_peaks(*model*, *qstar=None*, *max_order=4*)

class AFL.automation.shared.DiffractionLabeler.**DiffractionLabelerView**

__init__()

update_plot(*x*, *y*, *composition*)

remove_vertical_lines()

add_vertical_line(*x*, *y0=0*, *y1=128*, *row=1*, *col=1*, *line_kw=None*)

update_ternary_colors(*colors*)

run(*x*, *y*, *all_compositions*, *composition*, *models*, *possible_phase_labels*)

AFL.automation.shared.MutableQueue

Classes

MutableQueue()	Thread-safe, mutable queue

AFL.automation.shared.MutableQueue.MutableQueue

class AFL.automation.shared.MutableQueue.**MutableQueue**

Bases: object

Thread-safe, mutable queue

Unlike the standard library CPython queue, this class supportes positional inserts, deletions, and reordering. The tradeoff is performance for both reads and writes to the queue.

__init__()

Methods

__init__()	
empty()	
get([loc, block, timeout])	Get next item from queue
iterationid()	
move(old_index[, new_index])	Move item in queue
put(item, loc)	Insert an item at the top of the queue
qsize()	
remove(loc)	Remove an item from the queue

__init__()

qsize()

iterationid()

empty()

put(*item*, *loc*)

Insert an item at the top of the queue

remove(*loc*)

Remove an item from the queue

get(*loc=0*, *block=True*, *timeout=None*)

Get next item from queue

move(*old_index*, *new_index=None*)

Move item in queue

Exceptions

Empty	Exception raised by Queue.get(block=0)/get_nowait().
Full	Exception raised by Queue.put(block=0)/put_nowait().

AFL.automation.shared.MutableQueue.Empty

exception AFL.automation.shared.MutableQueue.**Empty**

Exception raised by Queue.get(block=0)/get_nowait().

AFL.automation.shared.MutableQueue.Full

exception AFL.automation.shared.MutableQueue.**Full**

Exception raised by Queue.put(block=0)/put_nowait().

class AFL.automation.shared.MutableQueue.**MutableQueue**

Thread-safe, mutable queue

Unlike the standard library CPython queue, this class supportes positional inserts, deletions, and reordering. The tradeoff is performance for both reads and writes to the queue.

__init__()

qsize()

iterationid()

empty()

put(*item*, *loc*)

Insert an item at the top of the queue

remove(*loc*)

Remove an item from the queue

get(*loc=0*, *block=True*, *timeout=None*)

Get next item from queue

move(*old_index*, *new_index=None*)

Move item in queue

AFL.automation.shared.PersistentConfig

Classes

MutableMapping()	A MutableMapping is a generic container for associating key/value pairs.
PersistentConfig(path[, defaults, ...])	A dictionary-like class that serializes changes to disk

AFL.automation.shared.PersistentConfig.MutableMapping

class AFL.automation.shared.PersistentConfig.**MutableMapping**

Bases: Mapping

A MutableMapping is a generic container for associating key/value pairs.

This class provides concrete generic implementations of all methods except for __getitem__, __setitem__, __delitem__, __iter__, and __len__.

__init__()

Methods

__init__()	
clear()	
get(k[,d])	
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

AFL.automation.shared.PersistentConfig.PersistentConfig

class AFL.automation.shared.PersistentConfig.**PersistentConfig**(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Bases: *MutableMapping*

A dictionary-like class that serializes changes to disk

This class provides dictionary-like setters and getters (e.g., [] and .update()) but, by default, all modifications are written into a file in json format with the root keys as timestamps. Modifications can be blocked by locking the config, and writing to disk can be disabled by setting the appropriate member attributes (see constructor). On instantiation, if provided with a previously saved configuration file, PersistentConfig will load the file's contents into memory and use the more recent configuration.

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

Methods

__init__(path[, defaults, overrides, lock, ...])	Constructor
clear()	
get(k[,d])	
get_historical_values(key[, convert_to_datetime])	Convenience method for gathering historical values of a parameter
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
revert([nth, datetime_key])	Revert config to a historical config
setdefault(k[,d])	
toJSON()	Serialize the config to json
update(update_dict)	Update several values in config at once
values()	

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

__getitem__(*key*)

Dictionary-like getter via config["param"]

__setitem__(*key*, *value*)

Dictionary-like setter via config["param"]=value

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

toJSON()

Serialize the config to json

update(*update_dict*)

Update several values in config at once

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

revert(*nth=None*, *datetime_key=None*)

Revert config to a historical config

Parameters

- **nth** (int, **optional***) – Integer index of historical value to revert to. Can be negative to count from end of history. Note that -1 will correspond to the current config.
- **datetime_key** (str, **optional**) – datetime formatted string as defined by datetime_key_format

get_historical_values(*key*, *convert_to_datetime=False*)

Convenience method for gathering historical values of a parameter

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

values() → an object providing a view on D's values

class AFL.automation.shared.PersistentConfig.**PersistentConfig**(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

A dictionary-like class that serializes changes to disk

This class provides dictionary-like setters and getters (e.g., [] and .update()) but, by default, all modifications are written into a file in json format with the root keys as timestamps. Modifications can be blocked by locking the config, and writing to disk can be disabled by setting the appropriate member attributes (see constructor). On instantiation, if provided with a previously saved configuration file, PersistentConfig will load the file's contents into memory and use the more recent configuration.

__init__(*path*, *defaults=None*, *overrides=None*, *lock=False*, *write=True*, *max_history=10000*, *datetime_key_format='%y/%d/%m %H:%M:%S.%f'*)

Constructor

Parameters

- **path** (*str or pathlib.Path*) – File path to file in which this config is or will be stored. This file will be created if it does not exist
- **defaults** (*dict*) – Default values to use if no saved config is available or if parameters are missing from a saved config.
- **overrides** (*dict*) – Values to use that override parameters in a saved config. These parameters can be changed after the PersistentConfig object is instantiated.
- **lock** (*bool*) – If True, an AttributeError will be raised if the user attempts to modify the config
- **write** (*bool*) – If False, all writing to the config file will be disabled and a warning will be emitted each time the config is modified.
- **datetime_key_format** (*str*) – String defining the root level keys of the json-serialized file. See https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

__getitem__(*key*)

Dictionary-like getter via config["param"]

__setitem__(*key*, *value*)

Dictionary-like setter via config["param"]=value

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

toJSON()

Serialize the config to json

update(*update_dict*)

Update several values in config at once

Changes will be written to PersistentConfig.path if PersistentConfig.write is True (default).

revert(*nth=None*, *datetime_key=None*)

Revert config to a historical config

Parameters

- **nth** (int, **optional***) – Integer index of historical value to revert to. Can be negative to count from end of history. Note that -1 will correspond to the current config.
- **datetime_key** (str, **optional**) – datetime formatted string as defined by datetime_key_format

get_historical_values(*key*, *convert_to_datetime=False*)

Convenience method for gathering historical values of a parameter

AFL.automation.shared.ServerDiscovery

Classes

AsyncServiceBrowser(zeroconf, type_[, ...])	Used to browse for a service for specific type(s).
AsyncServiceInfo	An async version of ServiceInfo.
AsyncZeroconf([interfaces, unicast, ...])	Implementation of Zeroconf Multicast DNS Service Discovery
AsyncZeroconfServiceTypes()	An async version of ZeroconfServiceTypes.
IPVersion(value[, names, module, qualname, ...])	
RunThread(coro)	
ServerDiscovery()	ServerDiscovery class
ServiceBrowser(zc, type_[, handlers, ...])	Used to browse for a service of a specific type.
ServiceInfo	Service information.
ServiceStateChange(value[, names, module, ...])	
Zeroconf([interfaces, unicast, ip_version, ...])	Implementation of Zeroconf Multicast DNS Service Discovery

AFL.automation.shared.ServerDiscovery.AsyncServiceBrowser

class AFL.automation.shared.ServerDiscovery.**AsyncServiceBrowser**(*zeroconf: Zeroconf*, *type_: str | list*, *handlers: ServiceListener | list[Callable[[...], None]] | None = None*, *listener: ServiceListener | None = None*, *addr: str | None = None*, *port: int = 5353*, *delay: int = 10000*, *question_type: DNSQuestionType | None = None*)

Bases: _ServiceBrowserBase

Used to browse for a service for specific type(s).

Constructor parameters are as follows:

- *zc*: A Zeroconf instance
- *type_*: fully qualified service type name
- *handler*: ServiceListener or Callable that knows how to process ServiceStateChange events
- *listener*: ServiceListener
- *addr*: address to send queries (will default to multicast)
- *port*: port to send queries (will default to mdns 5353)
- *delay*: The initial delay between answering questions
- *question_type*: The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

__init__(*zeroconf: Zeroconf*, *type_: str | list*, *handlers: ServiceListener | list[Callable[[...], None]] | None = None*, *listener: ServiceListener | None = None*, *addr: str | None = None*, *port: int = 5353*, *delay: int = 10000*, *question_type: DNSQuestionType | None = None*) → None

Used to browse for a service for specific type(s).

Constructor parameters are as follows:

- *zc*: A Zeroconf instance
- *type_*: fully qualified service type name
- *handler*: ServiceListener or Callable that knows how to process ServiceStateChange events
- *listener*: ServiceListener
- *addr*: address to send queries (will default to multicast)
- *port*: port to send queries (will default to mdns 5353)
- *delay*: The initial delay between answering questions
- *question_type*: The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

Methods

__init__(zeroconf, type_[, handlers, ...])	Used to browse for a service for specific type(s).
async_cancel()	Cancel the browser.
async_update_records(zc, now, records)	Callback invoked by Zeroconf when new information arrives.
async_update_records_complete()	Called when a record update has completed for all handlers.
update_record(zc, now, record)	Update a single record.

Attributes

done
query_scheduler
types
zc
service_state_changed

__init__(*zeroconf: Zeroconf*, *type_: str | list*, *handlers: ServiceListener | list[Callable[[...], None]] | None = None*, *listener: ServiceListener | None = None*, *addr: str | None = None*, *port: int = 5353*, *delay: int = 10000*, *question_type: DNSQuestionType | None = None*) → None

Used to browse for a service for specific type(s).

Constructor parameters are as follows:

- *zc*: A Zeroconf instance
- *type_*: fully qualified service type name

- *handler*: ServiceListener or Callable that knows how to process ServiceStateChange events
- *listener*: ServiceListener
- *addr*: address to send queries (will default to multicast)
- *port*: port to send queries (will default to mdns 5353)
- *delay*: The initial delay between answering questions
- *question_type*: The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

async async_cancel() → None

Cancel the browser.

done

query_scheduler

types

zc

async_update_records(*zc:* Zeroconf, *now: float_*, *records: list[RecordUpdate]*)

Callback invoked by Zeroconf when new information arrives.

Updates information required by browser in the Zeroconf cache.

Ensures that there is are no unnecessary duplicates in the list.

This method will be run in the event loop.

async_update_records_complete()

Called when a record update has completed for all handlers.

At this point the cache will have the new records.

This method will be run in the event loop.

This method is expected to be overridden by subclasses.

service_state_changed

update_record(*zc:* Zeroconf, *now: float*, *record: DNSRecord*) → None

Update a single record.

This method is deprecated and will be removed in a future version. update_records should be implemented instead.

AFL.automation.shared.ServerDiscovery.AsyncServiceInfo

class AFL.automation.shared.ServerDiscovery.**AsyncServiceInfo**

Bases: *ServiceInfo*

An async version of ServiceInfo.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
addresses_by_version(version)	List addresses matching IP version.
async_clear_cache()	Clear the cache for this service info.
async_request(zc, timeout[, question_type, ...])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
async_update_records(zc, now, records)	Updates service information from a DNS record.
async_update_records_complete()	Called when a record update has completed for all handlers.
async_wait(timeout[, loop])	Calling task waits for a given number of milliseconds or until notified.
dns_addresses([override_ttl, version])	Return matching DNSAddress from ServiceInfo.
dns_nsec(missing_types[, override_ttl])	Return DNSNsec from ServiceInfo.
dns_pointer([override_ttl])	Return DNSPointer from ServiceInfo.
dns_service([override_ttl])	Return DNSService from ServiceInfo.
dns_text([override_ttl])	Return DNSText from ServiceInfo.
get_address_and_nsec_records([override_ttl])	Build a set of address records and NSEC records for non-present record types.
get_name()	Name accessor
ip_addresses_by_version(version)	List ip_address objects matching IP version.
load_from_cache(zc[, now])	Populate the service info from the cache.
parsed_addresses([version])	List addresses in their parsed string form.
parsed_scoped_addresses([version])	Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available
request(zc, timeout[, question_type, addr, port])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
set_server_if_missing()	Set the server if it is missing.
update_record(zc, now, record)	Update a single record.

Attributes

host_ttl	
interface_index	
key	
other_ttl	
port	
priority	
server	
server_key	
text	
type	
weight	
addresses	IPv4 addresses of this service.
decoded_properties	Return properties as strings.
name	The name of the service.
properties	Return properties as bytes.

host_ttl

interface_index

key

other_ttl

port

priority

server

server_key

text

type

weight

__init__(**args*, ***kwargs*)

addresses

IPv4 addresses of this service.

Only IPv4 addresses are returned for backward compatibility. Use *addresses_by_version()* or *parsed_addresses()* to include IPv6 addresses as well.

addresses_by_version(*version:* IPVersion)

List addresses matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

async_clear_cache()

Clear the cache for this service info.

async async_request(*zc:* Zeroconf, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

This method will be run in the event loop.

Passing addr and port is optional, and will default to the mDNS multicast address and port. This is useful for directing requests to a specific host that may be able to respond across subnets.

Parameters

- **zc** – Zeroconf instance
- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

async_update_records(*zc:* Zeroconf, *now: float_*, *records: list[RecordUpdate]*)

Updates service information from a DNS record.

This method will be run in the event loop.

async_update_records_complete()

Called when a record update has completed for all handlers.

At this point the cache will have the new records.

This method will be run in the event loop.

async async_wait(*timeout: float*, *loop: AbstractEventLoop | None = None*) → None

Calling task waits for a given number of milliseconds or until notified.

decoded_properties

Return properties as strings.

dns_addresses(*override_ttl: int_ | None = None*, *version:* IPVersion *= IPVersion.All*) → list[DNSAddress]

Return matching DNSAddress from ServiceInfo.

dns_nsec(*missing_types: list[int]*, *override_ttl: int_ | None = None*) → DNSNsec

Return DNSNsec from ServiceInfo.

dns_pointer(*override_ttl: int_ | None = None*) → DNSPointer

Return DNSPointer from ServiceInfo.

dns_service(*override_ttl: int_ | None = None*) → DNSService

Return DNSService from ServiceInfo.

dns_text(*override_ttl: int_ | None = None*) → DNSText

Return DNSText from ServiceInfo.

get_address_and_nsec_records(*override_ttl: int_ | None = None*) → set[DNSRecord]

Build a set of address records and NSEC records for non-present record types.

get_name() → str

Name accessor

ip_addresses_by_version(*version: IPVersion*)

List ip_address objects matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

load_from_cache(*zc: Zeroconf*, *now: float_ | None = None*) → bool

Populate the service info from the cache.

This method is designed to be threadsafe.

name

The name of the service.

parsed_addresses(*version: IPVersion = IPVersion.All*) → list[str]

List addresses in their parsed string form.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

parsed_scoped_addresses(*version: IPVersion = IPVersion.All*) → list[str]

Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

properties

Return properties as bytes.

request(*zc: Zeroconf*, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_request* cannot be completed.

Parameters

- **zc** – Zeroconf instance

- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

set_server_if_missing() → None

Set the server if it is missing.

This function is for backwards compatibility.

update_record(*zc:* Zeroconf, *now: float*, *record: DNSRecord*) → None

Update a single record.

This method is deprecated and will be removed in a future version. update_records should be implemented instead.

AFL.automation.shared.ServerDiscovery.AsyncZeroconf

class AFL.automation.shared.ServerDiscovery.**AsyncZeroconf**(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version:* IPVersion | *None = None*, *apple_p2p: bool = False*, *zc:* Zeroconf | *None = None*)

Bases: object

Implementation of Zeroconf Multicast DNS Service Discovery

Supports registration, unregistration, queries and browsing.

The async version is currently a wrapper around Zeroconf which is now also async. It is expected that an asyncio event loop is already running before creating the AsyncZeroconf object.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version:* IPVersion | *None = None*, *apple_p2p: bool = False*, *zc:* Zeroconf | *None = None*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, and listening.

Parameters

- **interfaces** – InterfaceChoice or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (::1) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.
- **apple_p2p** – use AWDL interface (only macOS)

Methods

__init__([interfaces, unicast, ip_version, ...])	Creates an instance of the Zeroconf class, establishing multicast communications, and listening.
async_add_service_listener(type_, listener)	Adds a listener for a particular service type.
async_close()	Ends the background threads, and prevent this instance from servicing further queries.
async_get_service_info(type_, name[, ...])	Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.
async_register_service(info[, ttl, ...])	Registers service information to the network with a default TTL.
async_remove_all_service_listeners()	Removes a listener from the set that is currently listening.
async_remove_service_listener(listener)	Removes a listener from the set that is currently listening.
async_unregister_all_services()	Unregister all registered services.
async_unregister_service(info)	Unregister a service.
async_update_service(info)	Registers service information to the network with a default TTL.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version: IPVersion | None = None*, *apple_p2p: bool = False*, *zc: Zeroconf | None = None*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, and listening.

Parameters

- **interfaces** – `InterfaceChoice` or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (`::1`) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.

- **apple_p2p** – use AWDL interface (only macOS)

async async_register_service(*info: ServiceInfo*, *ttl: int | None = None*, *allow_name_change: bool = False*, *cooperating_responders: bool = False*, *strict: bool = True*) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service. The name of the service may be changed if needed to make it unique on the network. Additionally multiple cooperating responders can register the same service on the network for resilience (if you want this behavior set *cooperating_responders* to *True*).

The service will be broadcast in a task. This task is returned and therefore can be awaited if necessary.

async async_unregister_all_services() → None

Unregister all registered services.

Unlike async_register_service and async_unregister_service, this method does not return a future and is always expected to be awaited since its only called at shutdown.

async async_unregister_service(*info:* ServiceInfo) → Awaitable

Unregister a service.

The service will be broadcast in a task. This task is returned and therefore can be awaited if necessary.

async async_update_service(*info:* ServiceInfo) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service.

The service will be broadcast in a task. This task is returned and therefore can be awaited if necessary.

async async_close() → None

Ends the background threads, and prevent this instance from servicing further queries.

async async_get_service_info(*type_: str*, *name: str*, *timeout: int = 3000*, *question_type: DNSQuestionType | None = None*) → *AsyncServiceInfo* | None

Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.

Parameters

- **type** – fully qualified service type name
- **name** – the name of the service
- **timeout** – milliseconds to wait for a response
- **question_type** – The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

async async_add_service_listener(*type_: str*, *listener: ServiceListener*) → None

Adds a listener for a particular service type. This object will then have its add_service and remove_service methods called when services of that type become available and unavailable.

async async_remove_service_listener(*listener: ServiceListener*) → None

Removes a listener from the set that is currently listening.

async async_remove_all_service_listeners() → None

Removes a listener from the set that is currently listening.

AFL.automation.shared.ServerDiscovery.AsyncZeroconfServiceTypes

class AFL.automation.shared.ServerDiscovery.**AsyncZeroconfServiceTypes**

Bases: ZeroconfServiceTypes

An async version of ZeroconfServiceTypes.

__init__() → None

Keep track of found services in a set.

Methods

__init__()	Keep track of found services in a set.
add_service(zc, type_, name)	Service added.
async_find([aiozc, timeout, interfaces, ...])	Return all of the advertised services on any local networks.
find([zc, timeout, interfaces, ip_version])	Return all of the advertised services on any local networks.
remove_service(zc, type_, name)	Service removed.
update_service(zc, type_, name)	Service updated.

async classmethod async_find(*aiozc:* AsyncZeroconf | *None* = *None*, *timeout:* *int* | *float* = *5*, *interfaces:* *Sequence[str* | *int* | *tuple[tuple[str, int, int], int]]* | *InterfaceChoice* = *InterfaceChoice.All*, *ip_version:* IPVersion | *None* = *None*) → tuple[str, ...]

Return all of the advertised services on any local networks.

Parameters

- **aiozc** – AsyncZeroconf() instance. Pass in if already have an instance running or if non-default interfaces are needed
- **timeout** – seconds to wait for any responses
- **interfaces** – interfaces to listen on.
- **ip_version** – IP protocol version to use.

Returns

tuple of service type strings

__init__() → None

Keep track of found services in a set.

add_service(*zc:* Zeroconf, *type_:* *str*, *name:* *str*) → None

Service added.

classmethod find(*zc:* Zeroconf | *None* = *None*, *timeout:* *int* | *float* = *5*, *interfaces:* *Sequence[str* | *int* | *tuple[tuple[str, int, int], int]]* | *InterfaceChoice* = *InterfaceChoice.All*, *ip_version:* IPVersion | *None* = *None*) → tuple[str, ...]

Return all of the advertised services on any local networks.

Parameters

- **zc** – Zeroconf() instance. Pass in if already have an instance running or if non-default interfaces are needed
- **timeout** – seconds to wait for any responses
- **interfaces** – interfaces to listen on.
- **ip_version** – IP protocol version to use.

Returns

tuple of service type strings

remove_service(*zc:* Zeroconf, *type_:* *str*, *name:* *str*) → None

Service removed.

update_service(*zc:* Zeroconf, *type_: str*, *name: str*) → None

Service updated.

AFL.automation.shared.ServerDiscovery.IPVersion

class AFL.automation.shared.ServerDiscovery.**IPVersion**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *Enum*

__init__(**args*, ***kwds*)

Attributes

V4Only
V6Only
All

V4Only = 1

V6Only = 2

All = 3

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

AFL.automation.shared.ServerDiscovery.RunThread

class AFL.automation.shared.ServerDiscovery.**RunThread**(*coro*)

Bases: Thread

__init__(*coro*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(coro)	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*coro*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

AFL.automation.shared.ServerDiscovery.ServerDiscovery

class AFL.automation.shared.ServerDiscovery.**ServerDiscovery**

Bases: object

ServerDiscovery class

This class is used to discover AFL servers on the network using zeroconf requests that match a particular specification.

__init__()

Methods

__init__()	
aio_find_server_by_name(service_name)	
discover_server_by_name(service_name)	Does a zeroconf request to find a named AFL-automation server on the network.
find_server_by_name(service_name)	Disambiguator for either matching or discovering a specific server by name
find_server_by_partial_name(service_name)	Looks through the registry of discovered services for a partial name match.
find_server_by_property_match(property_name ...)	Looks through the registry of discovered services for a partial match in a property string.
get_service_info(zeroconf, service_type, name)	
manage_service_info_to_list(zeroconf, ...)	
match_server_by_name(service_name)	Looks through the registry of discovered services for an exact name match.
on_service_state_change(zeroconf, ...)	
sa_aio_discover_server_by_name(service_name	Does a zeroconf request to find a named AFL-automation server on the network.
sa_discover_server_by_name(service_name)	Does a zeroconf request to find a named AFL-automation server on the network.

__init__()

on_service_state_change(*zeroconf: Zeroconf*, *service_type: str*, *name: str*, *state_change: ServiceStateChange*) → None

async manage_service_info_to_list(*zeroconf*, *service_type*, *name*, *append_or_remove*)

async get_service_info(*zeroconf: Zeroconf*, *service_type: str*, *name: str*) → None

async aio_find_server_by_name(*service_name*)

discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

async classmethod sa_aio_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

classmethod sa_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_name(*service_name*)

Disambiguator for either matching or discovering a specific server by name

match_server_by_name(*service_name*)

Looks through the registry of discovered services for an exact name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_partial_name(*service_name*)

Looks through the registry of discovered services for a partial name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_property_match(*property_name*, *property_value*)

Looks through the registry of discovered services for a partial match in a property string. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

AFL.automation.shared.ServerDiscovery.ServiceBrowser

class AFL.automation.shared.ServerDiscovery.**ServiceBrowser**(*zc: Zeroconf*, *type_: str | list*, *handlers: ServiceListener | list[Callable[..., None]] | None = None*, *listener: ServiceListener | None = None*, *addr: str | None = None*, *port: int = 5353*, *delay: int = 10000*, *question_type: DNSQuestionType | None = None*)

Bases: _ServiceBrowserBase, Thread

Used to browse for a service of a specific type.

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

__init__(*zc: Zeroconf*, *type_: str | list*, *handlers: ServiceListener | list[Callable[..., None]] | None = None*, *listener: ServiceListener | None = None*, *addr: str | None = None*, *port: int = 5353*, *delay: int = 10000*, *question_type: DNSQuestionType | None = None*) → None

Used to browse for a service for specific type(s).

Constructor parameters are as follows:

- *zc*: A Zeroconf instance
- *type_*: fully qualified service type name
- *handler*: ServiceListener or Callable that knows how to process ServiceStateChange events
- *listener*: ServiceListener
- *addr*: address to send queries (will default to multicast)
- *port*: port to send queries (will default to mdns 5353)
- *delay*: The initial delay between answering questions
- *question_type*: The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

Methods

__init__(zc, type_[, handlers, listener, ...])	Used to browse for a service for specific type(s).
async_update_records(zc, now, records)	Callback invoked by Zeroconf when new information arrives.
async_update_records_complete()	Called when a record update has completed for all handlers.
cancel()	Cancel the browser.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
run()	Run the browser thread.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
update_record(zc, now, record)	Update a single record.

Attributes

done	
query_scheduler	
types	
zc	
daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.
service_state_changed	

done

query_scheduler

types

zc

__init__(*zc:* Zeroconf, *type_:* *str* | *list*, *handlers:* *ServiceListener* | *list[Callable[..., None]]* | *None* = *None*, *listener:* *ServiceListener* | *None* = *None*, *addr:* *str* | *None* = *None*, *port:* *int* = *5353*, *delay:* *int* = *10000*, *question_type:* *DNSQuestionType* | *None* = *None*) → None

Used to browse for a service for specific type(s).

Constructor parameters are as follows:

- *zc*: A Zeroconf instance
- *type_*: fully qualified service type name
- *handler*: ServiceListener or Callable that knows how to process ServiceStateChange events
- *listener*: ServiceListener
- *addr*: address to send queries (will default to multicast)
- *port*: port to send queries (will default to mdns 5353)
- *delay*: The initial delay between answering questions
- *question_type*: The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

The listener object will have its add_service() and remove_service() methods called when this browser discovers changes in the services availability.

async_update_records(*zc:* Zeroconf, *now:* *float_*, *records:* *list[RecordUpdate]*)

Callback invoked by Zeroconf when new information arrives.

Updates information required by browser in the Zeroconf cache.

Ensures that there is are no unnecessary duplicates in the list.

This method will be run in the event loop.

async_update_records_complete() → None

Called when a record update has completed for all handlers.

At this point the cache will have the new records.

This method will be run in the event loop.

cancel() → None

Cancel the browser.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run() → None

Run the browser thread.

service_state_changed

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

update_record(*zc:* Zeroconf, *now: float*, *record: DNSRecord*) → None

Update a single record.

This method is deprecated and will be removed in a future version. update_records should be implemented instead.

AFL.automation.shared.ServerDiscovery.ServiceInfo

class AFL.automation.shared.ServerDiscovery.**ServiceInfo**

Bases: RecordUpdateListener

Service information.

Constructor parameters are as follows:

- *type_*: fully qualified service type name
- *name*: fully qualified service name
- *port*: port that the service runs on
- *weight*: weight of the service
- *priority*: priority of the service
- *properties*: dictionary of properties (or a bytes object holding the contents of the *text* field). converted to str and then encoded to bytes using UTF-8. Keys with *None* values are converted to value-less attributes.
- *server*: fully qualified name for service host (defaults to name)

- *host_ttl*: ttl used for A/SRV records
- *other_ttl*: ttl used for PTR/TXT records
- *addresses* and *parsed_addresses*: List of IP addresses (either as bytes, network byte order, or in parsed form as text; at most one of those parameters can be provided)
- interface_index: scope_id or zone_id for IPv6 link-local addresses i.e. an identifier of the interface where the peer is connected to

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
addresses_by_version(version)	List addresses matching IP version.
async_clear_cache()	Clear the cache for this service info.
async_request(zc, timeout[, question_type, ...])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
async_update_records(zc, now, records)	Updates service information from a DNS record.
async_update_records_complete()	Called when a record update has completed for all handlers.
async_wait(timeout[, loop])	Calling task waits for a given number of milliseconds or until notified.
dns_addresses([override_ttl, version])	Return matching DNSAddress from ServiceInfo.
dns_nsec(missing_types[, override_ttl])	Return DNSNsec from ServiceInfo.
dns_pointer([override_ttl])	Return DNSPointer from ServiceInfo.
dns_service([override_ttl])	Return DNSService from ServiceInfo.
dns_text([override_ttl])	Return DNSText from ServiceInfo.
get_address_and_nsec_records([override_ttl])	Build a set of address records and NSEC records for non-present record types.
get_name()	Name accessor
ip_addresses_by_version(version)	List ip_address objects matching IP version.
load_from_cache(zc[, now])	Populate the service info from the cache.
parsed_addresses([version])	List addresses in their parsed string form.
parsed_scoped_addresses([version])	Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available
request(zc, timeout[, question_type, addr, port])	Returns true if the service could be discovered on the network, and updates this object with details discovered.
set_server_if_missing()	Set the server if it is missing.
update_record(zc, now, record)	Update a single record.

Attributes

host_ttl	
interface_index	
key	
other_ttl	
port	
priority	
server	
server_key	
text	
type	
weight	
addresses	IPv4 addresses of this service.
decoded_properties	Return properties as strings.
name	The name of the service.
properties	Return properties as bytes.

host_ttl

interface_index

key

other_ttl

port

priority

server

server_key

text

type

weight

__init__(**args*, ***kwargs*)

addresses

IPv4 addresses of this service.

Only IPv4 addresses are returned for backward compatibility. Use *addresses_by_version()* or *parsed_addresses()* to include IPv6 addresses as well.

addresses_by_version(*version:* IPVersion)

List addresses matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

async_clear_cache()

Clear the cache for this service info.

async async_request(*zc:* Zeroconf, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

This method will be run in the event loop.

Passing addr and port is optional, and will default to the mDNS multicast address and port. This is useful for directing requests to a specific host that may be able to respond across subnets.

Parameters

- **zc** – Zeroconf instance
- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

async_update_records(*zc:* Zeroconf, *now: float_*, *records: list[RecordUpdate]*)

Updates service information from a DNS record.

This method will be run in the event loop.

async_update_records_complete()

Called when a record update has completed for all handlers.

At this point the cache will have the new records.

This method will be run in the event loop.

async async_wait(*timeout: float*, *loop: AbstractEventLoop | None = None*) → None

Calling task waits for a given number of milliseconds or until notified.

decoded_properties

Return properties as strings.

dns_addresses(*override_ttl: int_ | None = None*, *version:* IPVersion *= IPVersion.All*) → list[DNSAddress]

Return matching DNSAddress from ServiceInfo.

dns_nsec(*missing_types: list[int]*, *override_ttl: int_ | None = None*) → DNSNsec

Return DNSNsec from ServiceInfo.

dns_pointer(*override_ttl: int_ | None = None*) → DNSPointer

Return DNSPointer from ServiceInfo.

dns_service(*override_ttl: int_ | None = None*) → DNSService

Return DNSService from ServiceInfo.

dns_text(*override_ttl: int_ | None = None*) → DNSText

Return DNSText from ServiceInfo.

get_address_and_nsec_records(*override_ttl: int_ | None = None*) → set[DNSRecord]

Build a set of address records and NSEC records for non-present record types.

get_name() → str

Name accessor

ip_addresses_by_version(*version: IPVersion*)

List ip_address objects matching IP version.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

load_from_cache(*zc: Zeroconf*, *now: float_ | None = None*) → bool

Populate the service info from the cache.

This method is designed to be threadsafe.

name

The name of the service.

parsed_addresses(*version: IPVersion = IPVersion.All*) → list[str]

List addresses in their parsed string form.

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

parsed_scoped_addresses(*version: IPVersion = IPVersion.All*) → list[str]

Equivalent to parsed_addresses, with the exception that IPv6 Link-Local addresses are qualified with %<interface_index> when available

Addresses are guaranteed to be returned in LIFO (last in, first out) order with IPv4 addresses first and IPv6 addresses second.

This means the first address will always be the most recently added address of the given IP version.

properties

Return properties as bytes.

request(*zc: Zeroconf*, *timeout: float*, *question_type: DNSQuestionType | None = None*, *addr: str | None = None*, *port: int = 5353*) → bool

Returns true if the service could be discovered on the network, and updates this object with details discovered.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_request* cannot be completed.

Parameters

- **zc** – Zeroconf instance

- **timeout** – time in milliseconds to wait for a response
- **question_type** – question type to ask
- **addr** – address to send the request to
- **port** – port to send the request to

set_server_if_missing() → None

Set the server if it is missing.

This function is for backwards compatibility.

update_record(*zc:* Zeroconf, *now: float*, *record: DNSRecord*) → None

Update a single record.

This method is deprecated and will be removed in a future version. update_records should be implemented instead.

AFL.automation.shared.ServerDiscovery.ServiceStateChange

class AFL.automation.shared.ServerDiscovery.**ServiceStateChange**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *Enum*

__init__(**args*, ***kwds*)

Attributes

Added
Removed
Updated

Added = 1

Removed = 2

Updated = 3

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

classmethod __getitem__(*name*)

Return the member matching *name*.

classmethod __iter__()

Return members in definition order.

classmethod __len__()

Return the number of members (no aliases)

AFL.automation.shared.ServerDiscovery.Zeroconf

class AFL.automation.shared.ServerDiscovery.**Zeroconf**(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version: IPVersion | None = None*, *apple_p2p: bool = False*)

Bases: `QuietLogger`

Implementation of Zeroconf Multicast DNS Service Discovery

Supports registration, unregistration, queries and browsing.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version: IPVersion | None = None*, *apple_p2p: bool = False*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.

Parameters

- **interfaces** – `InterfaceChoice` or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (`::1`) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.

- **apple_p2p** – use AWDL interface (only macOS)

Methods

__init__([interfaces, unicast, ip_version, ...])	Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.
add_listener(listener, question)	Adds a listener for a given question.
add_service_listener(type_, listener)	Adds a listener for a particular service type.
async_add_listener(listener, question)	Adds a listener for a given question.
async_check_service(info, allow_name_change)	Checks the network for a unique service name, modifying the ServiceInfo passed in if it is not unique.
async_get_service_info(type_, name[, ...])	Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.
async_notify_all()	Schedule an async_notify_all.
async_register_service(info[, ttl, ...])	Registers service information to the network with a default TTL.
async_remove_listener(listener)	Removes a listener.
async_send(out[, addr, port, v6_flow_scope, ...])	Sends an outgoing packet.
async_unregister_all_services()	Unregister all registered services.
async_unregister_service(info)	Unregister a service.

continues on next page

Table 60 – continued from previous page

async_update_service(info)	Registers service information to the network with a default TTL.
async_wait(timeout)	Calling task waits for a given number of milliseconds or until notified.
async_wait_for_start([timeout])	Wait for start up for actions that require a running Zeroconf instance.
close()	Ends the background threads, and prevent this instance from servicing further queries.
generate_service_broadcast(info, ttl[, ...])	Generate a broadcast to announce a service.
generate_service_query(info)	Generate a query to lookup a service.
generate_unregister_all_services()	Generate a DNSOutgoing goodbye for all services and remove them from the registry.
get_service_info(type_, name[, timeout, ...])	Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.
log_exception_debug(*logger_data)	
log_exception_once(exc, *args)	
log_exception_warning(*logger_data)	
log_warning_once(*args)	
notify_all()	Notifies all waiting threads and notify listeners.
register_service(info[, ttl, ...])	Registers service information to the network with a default TTL.
remove_all_service_listeners()	Removes a listener from the set that is currently listening.
remove_listener(listener)	Removes a listener.
remove_service_listener(listener)	Removes a listener from the set that is currently listening.
send(out[, addr, port, v6_flow_scope, transport])	Sends an outgoing packet threadsafe.
start()	Start Zeroconf.
unregister_all_services()	Unregister all registered services.
unregister_service(info)	Unregister a service.
update_service(info)	Registers service information to the network with a default TTL.

Attributes

listeners	
started	Check if the instance has started.

__init__(*interfaces: Sequence[str | int | tuple[tuple[str, int, int], int]] | InterfaceChoice = InterfaceChoice.All*, *unicast: bool = False*, *ip_version:* IPVersion | *None = None*, *apple_p2p: bool = False*) → None

Creates an instance of the Zeroconf class, establishing multicast communications, listening and reaping threads.

Parameters

- **interfaces** – `InterfaceChoice` or a list of IP addresses (IPv4 and IPv6) and interface indexes (IPv6 only).

  IPv6 notes for non-POSIX systems: * *InterfaceChoice.All* is an alias for *InterfaceChoice.Default*

  on Python versions before 3.8.

  Also listening on loopback (`::1`) doesn't work, use a real address.

- **ip_version** – IP versions to support. If *choice* is a list, the default is detected from it. Otherwise defaults to V4 only for backward compatibility.
- **apple_p2p** – use AWDL interface (only macOS)

property started: bool

Check if the instance has started.

start() → None

Start Zeroconf.

async async_wait_for_start(*timeout: float = 9*) → None

Wait for start up for actions that require a running Zeroconf instance.

Throws NotRunningException if the instance is not running or could not be started.

property listeners: set[RecordUpdateListener]

async async_wait(*timeout: float*) → None

Calling task waits for a given number of milliseconds or until notified.

notify_all() → None

Notifies all waiting threads and notify listeners.

async_notify_all() → None

Schedule an async_notify_all.

get_service_info(*type_: str*, *name: str*, *timeout: int = 3000*, *question_type: DNSQuestionType | None = None*) → *ServiceInfo* | None

Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.

Parameters

- **type** – fully qualified service type name
- **name** – the name of the service
- **timeout** – milliseconds to wait for a response
- **question_type** – The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

add_service_listener(*type_: str*, *listener: ServiceListener*) → None

Adds a listener for a particular service type. This object will then have its add_service and remove_service methods called when services of that type become available and unavailable.

remove_service_listener(*listener: ServiceListener*) → None

Removes a listener from the set that is currently listening.

remove_all_service_listeners() → None

Removes a listener from the set that is currently listening.

register_service(*info:* ServiceInfo, *ttl:* *int* | *None* = *None*, *allow_name_change:* *bool* = *False*, *cooperating_responders:* *bool* = *False*, *strict:* *bool* = *True*) → None

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service. The name of the service may be changed if needed to make it unique on the network. Additionally multiple cooperating responders can register the same service on the network for resilience (if you want this behavior set *cooperating_responders* to *True*).

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *register_service* cannot be completed.

async async_register_service(*info:* ServiceInfo, *ttl:* *int* | *None* = *None*, *allow_name_change:* *bool* = *False*, *cooperating_responders:* *bool* = *False*, *strict:* *bool* = *True*) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service. The name of the service may be changed if needed to make it unique on the network. Additionally multiple cooperating responders can register the same service on the network for resilience (if you want this behavior set *cooperating_responders* to *True*).

update_service(*info:* ServiceInfo) → None

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_update_service* cannot be completed.

async async_update_service(*info:* ServiceInfo) → Awaitable

Registers service information to the network with a default TTL. Zeroconf will then respond to requests for information for that service.

async async_get_service_info(*type_:* *str*, *name:* *str*, *timeout:* *int* = *3000*, *question_type:* *DNSQuestionType* | *None* = *None*) → *AsyncServiceInfo* | None

Returns network's service information for a particular name and type, or None if no service matches by the timeout, which defaults to 3 seconds.

Parameters

- **type** – fully qualified service type name
- **name** – the name of the service
- **timeout** – milliseconds to wait for a response
- **question_type** – The type of questions to ask (DNSQuestionType.QM or DNSQuestionType.QU)

generate_service_broadcast(*info:* ServiceInfo, *ttl:* *int* | *None*, *broadcast_addresses:* *bool* = *True*) → DNSOutgoing

Generate a broadcast to announce a service.

generate_service_query(*info:* ServiceInfo) → DNSOutgoing

Generate a query to lookup a service.

unregister_service(*info:* ServiceInfo) → None

Unregister a service.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_unregister_service* cannot be completed.

async async_unregister_service(*info:* ServiceInfo) → Awaitable

Unregister a service.

generate_unregister_all_services() → DNSOutgoing | None

Generate a DNSOutgoing goodbye for all services and remove them from the registry.

async async_unregister_all_services() → None

Unregister all registered services.

Unlike async_register_service and async_unregister_service, this method does not return a future and is always expected to be awaited since its only called at shutdown.

unregister_all_services() → None

Unregister all registered services.

While it is not expected during normal operation, this function may raise EventLoopBlocked if the underlying call to *async_unregister_all_services* cannot be completed.

async async_check_service(*info:* ServiceInfo, *allow_name_change: bool*, *cooperating_responders: bool* = *False*, *strict: bool* = *True*) → None

Checks the network for a unique service name, modifying the ServiceInfo passed in if it is not unique.

add_listener(*listener: RecordUpdateListener*, *question: DNSQuestion | list[DNSQuestion] | None*) → None

Adds a listener for a given question. The listener will have its update_record method called when information is available to answer the question(s).

This function is threadsafe

remove_listener(*listener: RecordUpdateListener*) → None

Removes a listener.

This function is threadsafe

async_add_listener(*listener: RecordUpdateListener*, *question: DNSQuestion | list[DNSQuestion] | None*) → None

Adds a listener for a given question. The listener will have its update_record method called when information is available to answer the question(s).

This function is not threadsafe and must be called in the eventloop.

async_remove_listener(*listener: RecordUpdateListener*) → None

Removes a listener.

This function is not threadsafe and must be called in the eventloop.

send(*out: DNSOutgoing*, *addr: str | None* = *None*, *port: int* = *5353*, *v6_flow_scope: tuple[()] | tuple[int, int]* = *()*, *transport: _WrappedTransport | None* = *None*) → None

Sends an outgoing packet threadsafe.

async_send(*out: DNSOutgoing*, *addr: str | None* = *None*, *port: int* = *5353*, *v6_flow_scope: tuple[()] | tuple[int, int]* = *()*, *transport: _WrappedTransport | None* = *None*) → None

Sends an outgoing packet.

close() → None

Ends the background threads, and prevent this instance from servicing further queries.

This method is idempotent and irreversible.

classmethod log_exception_debug(**logger_data: Any*) → None

classmethod log_exception_once(*exc: Exception*, **args: Any*) → None

classmethod log_exception_warning(**logger_data: Any*) → None

classmethod log_warning_once(**args: Any*) → None

class AFL.automation.shared.ServerDiscovery.**RunThread**(*coro*)

__init__(*coro*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class AFL.automation.shared.ServerDiscovery.**ServerDiscovery**

ServerDiscovery class

This class is used to discover AFL servers on the network using zeroconf requests that match a particular specification.

__init__()

on_service_state_change(*zeroconf: Zeroconf*, *service_type: str*, *name: str*, *state_change: ServiceStateChange*) → None

async manage_service_info_to_list(*zeroconf*, *service_type*, *name*, *append_or_remove*)

async get_service_info(*zeroconf: Zeroconf*, *service_type: str*, *name: str*) → None

async aio_find_server_by_name(*service_name*)

discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

async classmethod sa_aio_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

classmethod sa_discover_server_by_name(*service_name*)

Does a zeroconf request to find a named AFL-automation server on the network.

Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_name(*service_name*)

Disambiguator for either matching or discovering a specific server by name

match_server_by_name(*service_name*)

Looks through the registry of discovered services for an exact name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_partial_name(*service_name*)

Looks through the registry of discovered services for a partial name match. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

find_server_by_property_match(*property_name*, *property_value*)

Looks through the registry of discovered services for a partial match in a property string. Returns a tuple of (hostname:port string,ServiceInfo object) if found, None if not found.

AFL.automation.shared.exceptions

Exceptions

EmptyException	Raised when a mixture runs out of mass or volume
MixingException	Raised when a mixture cannot be made
NoDeviceFoundException	Raised when no matching device can be found on the selected port
NotFoundError	
SerialCommsException	Raised when the system receives a serial response it can't parse, likely a garbled line

AFL.automation.shared.exceptions.EmptyException

exception AFL.automation.shared.exceptions.**EmptyException**

Raised when a mixture runs out of mass or volume

AFL.automation.shared.exceptions.MixingException

exception AFL.automation.shared.exceptions.**MixingException**

Raised when a mixture cannot be made

AFL.automation.shared.exceptions.NoDeviceFoundException

exception AFL.automation.shared.exceptions.**NoDeviceFoundException**

Raised when no matching device can be found on the selected port

AFL.automation.shared.exceptions.NotFoundError

exception AFL.automation.shared.exceptions.**NotFoundError**

AFL.automation.shared.exceptions.SerialCommsException

exception AFL.automation.shared.exceptions.**SerialCommsException**

Raised when the system receives a serial response it can't parse, likely a garbled line

exception AFL.automation.shared.exceptions.**EmptyException**

Raised when a mixture runs out of mass or volume

exception AFL.automation.shared.exceptions.**MixingException**

Raised when a mixture cannot be made

exception AFL.automation.shared.exceptions.**SerialCommsException**

Raised when the system receives a serial response it can't parse, likely a garbled line

exception AFL.automation.shared.exceptions.**NoDeviceFoundException**

Raised when no matching device can be found on the selected port

exception AFL.automation.shared.exceptions.**NotFoundError**

AFL.automation.shared.serialization

Functions

deserialize(pickled_str)
is_serialized(obj)
serialize(obj)

AFL.automation.shared.serialization.deserialize

AFL.automation.shared.serialization.**deserialize**(*pickled_str*)

AFL.automation.shared.serialization.is_serialized

AFL.automation.shared.serialization.**is_serialized**(*obj*)

AFL.automation.shared.serialization.serialize

AFL.automation.shared.serialization.**serialize**(*obj*)

Exceptions

UnpicklingError

AFL.automation.shared.serialization.UnpicklingError

exception AFL.automation.shared.serialization.**UnpicklingError**

AFL.automation.shared.serialization.**serialize**(*obj*)

AFL.automation.shared.serialization.**is_serialized**(*obj*)

AFL.automation.shared.serialization.**deserialize**(*pickled_str*)

AFL.automation.shared.units

Functions

enforce_units(value, unit_type)	Ensure that a number has units and convert to the default_units
get_unit_type(value)	
has_units(value)	
is_concentration(value)	
is_density(value)	
is_mass(value)	
is_molarity(value)	
is_volume(value)	

AFL.automation.shared.units.enforce_units

AFL.automation.shared.units.**enforce_units**(*value*, *unit_type*)

Ensure that a number has units and convert to the default_units

AFL.automation.shared.units.get_unit_type

AFL.automation.shared.units.**get_unit_type**(*value*)

AFL.automation.shared.units.has_units

AFL.automation.shared.units.**has_units**(*value*)

AFL.automation.shared.units.is_concentration

AFL.automation.shared.units.**is_concentration**(*value*)

AFL.automation.shared.units.is_density

AFL.automation.shared.units.**is_density**(*value*)

AFL.automation.shared.units.is_mass

AFL.automation.shared.units.**is_mass**(*value*)

AFL.automation.shared.units.is_molarity

AFL.automation.shared.units.**is_molarity**(*value*)

AFL.automation.shared.units.is_volume

AFL.automation.shared.units.**is_volume**(*value*)

AFL.automation.shared.units.**has_units**(*value*)

AFL.automation.shared.units.**is_volume**(*value*)

AFL.automation.shared.units.**is_molarity**(*value*)

AFL.automation.shared.units.**is_mass**(*value*)

AFL.automation.shared.units.**is_density**(*value*)

AFL.automation.shared.units.**is_concentration**(*value*)

AFL.automation.shared.units.**get_unit_type**(*value*)

AFL.automation.shared.units.**enforce_units**(*value*, *unit_type*)

Ensure that a number has units and convert to the default_units

AFL.automation.shared.utilities

Functions

has_units(value)	
listify(obj)	
makeRegistar()	
mpl_plot_to_bytes([fig, format])	
tprint(in_str)	
xarray_to_bytes(ds[, format])	

AFL.automation.shared.utilities.has_units

AFL.automation.shared.utilities.**has_units**(*value*)

AFL.automation.shared.utilities.listify

AFL.automation.shared.utilities.**listify**(*obj*)

AFL.automation.shared.utilities.makeRegistar

AFL.automation.shared.utilities.**makeRegistar**()

AFL.automation.shared.utilities.mpl_plot_to_bytes

AFL.automation.shared.utilities.**mpl_plot_to_bytes**(*fig=None*, *format='svg'*)

AFL.automation.shared.utilities.tprint

AFL.automation.shared.utilities.**tprint**(*in_str*)

AFL.automation.shared.utilities.xarray_to_bytes

AFL.automation.shared.utilities.**xarray_to_bytes**(*ds*, *format='svg'*)

AFL.automation.shared.utilities.**listify**(*obj*)

AFL.automation.shared.utilities.**tprint**(*in_str*)

AFL.automation.shared.utilities.**makeRegistar**()

AFL.automation.shared.utilities.**mpl_plot_to_bytes**(*fig=None*, *format='svg'*)

AFL.automation.shared.utilities.**xarray_to_bytes**(*ds*, *format='svg'*)

AFL.automation.shared.widgetui

Functions

clear_output([wait])	Clear the output of the current cell receiving output.
client_construct_ui(self[, return_ui, ...])	
display(*objs[, include, exclude, metadata, ...])	Display a Python object in all frontends.

AFL.automation.shared.widgetui.clear_output

AFL.automation.shared.widgetui.**clear_output**(*wait=False*)

Clear the output of the current cell receiving output.

Parameters

wait (*bool [default: false]*) – Wait to clear the output until new output is available to replace it.

AFL.automation.shared.widgetui.client_construct_ui

AFL.automation.shared.widgetui.**client_construct_ui**(*self*, *return_ui=False*, *display_ui=True*)

AFL.automation.shared.widgetui.display

AFL.automation.shared.widgetui.**display**(**objs*, *include=None*, *exclude=None*, *metadata=None*, *transient=None*, *display_id=None*, *raw=False*, *clear=False*, ***kwargs*)

Display a Python object in all frontends.

By default all representations will be computed and sent to the frontends. Frontends can decide which representation is used and how.

In terminal IPython this will be similar to using `print()`, for use in richer frontends see Jupyter notebook examples with rich display logic.

Parameters

- ***objs** (*object*) – The Python objects to display.
- **raw** (*bool, optional*) – Are the objects to be displayed already mimetype-keyed dicts of raw display data, or Python objects that need to be formatted before display? [default: False]
- **include** (*list, tuple or set, optional*) – A list of format type strings (MIME types) to include in the format data dict. If this is set *only* the format types included in this list will be computed.
- **exclude** (*list, tuple or set, optional*) – A list of format type strings (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.
- **metadata** (*dict, optional*) – A dictionary of metadata to associate with the output. mime-type keys in this dictionary will be associated with the individual representation formats, if they exist.
- **transient** (*dict, optional*) – A dictionary of transient data to associate with the output. Data in this dict should not be persisted to files (e.g. notebooks).
- **display_id** (*str, bool optional*) – Set an id for the display. This id can be used for updating this display area later via update_display. If given as *True*, generate a new *display_id*
- **clear** (*bool, optional*) – Should the output area be cleared before displaying anything? If True, this will wait for additional output before clearing. [default: False]
- ****kwargs** (*additional keyword-args, optional*) – Additional keyword-arguments are passed through to the display publisher.

Returns

handle – Returns a handle on updatable displays for use with `update_display()`, if *display_id* is given. Returns `None` if no *display_id* is given (default).

Return type

DisplayHandle

Examples

```
>>> class Json(object):
...     def __init__(self, json):
...         self.json = json
...     def _repr_pretty_(self, pp, cycle):
...         import json
...         pp.text(json.dumps(self.json, indent=2))
...     def __repr__(self):
...         return str(self.json)
...
```

```
>>> d = Json({1:2, 3: {4:5}})
```

```
>>> print(d)
{1: 2, 3: {4: 5}}
```

```
>>> display(d)
{
  "1": 2,
  "3": {
    "4": 5
  }
}
```

```
>>> def int_formatter(integer, pp, cycle):
...     pp.text('I'*integer)
```

```
>>> plain = get_ipython().display_formatter.formatters['text/plain']
>>> plain.for_type(int, int_formatter)
<function _repr_pprint at 0x...>
>>> display(7-5)
II
```

```
>>> del plain.type_printers[int]
>>> display(7-5)
2
```

See also

`update_display()`

Notes

In Python, objects can declare their textual representation using the *__repr__* method. IPython expands on this idea and allows objects to declare other, rich representations including:

- HTML
- JSON
- PNG
- JPEG
- SVG
- LaTeX

A single object can declare some or all of these representations; all are handled by IPython's display system.

The main idea of the first approach is that you have to implement special display methods when you define your class, one for each representation you want to use. Here is a list of the names of the special methods and the values they must return:

- *_repr_html_*: return raw HTML as a string, or a tuple (see below).
- *_repr_json_*: return a JSONable dict, or a tuple (see below).

- *_repr_jpeg_*: return raw JPEG data, or a tuple (see below).
- *_repr_png_*: return raw PNG data, or a tuple (see below).
- *_repr_svg_*: return raw SVG data as a string, or a tuple (see below).
- ***_repr_latex_*: return LaTeX commands in a string surrounded by "$",** or a tuple (see below).
- ***_repr_mimebundle_*: return a full mimebundle containing the mapping** from all mimetypes to data. Use this for any mime-type not listed above.

The above functions may also return the object's metadata alonside the data. If the metadata is available, the functions will return a tuple containing the data and metadata, in that order. If there is no metadata available, then the functions will return the data only.

When you are directly writing your own classes, you can adapt them for display in IPython by following the above approach. But in practice, you often need to work with existing classes that you can't easily modify.

You can refer to the documentation on integrating with the display system in order to register custom formatters for already existing types (integrating_rich_display).

Added in version 5.4: display available without import

Added in version 6.1: display available without import

Since IPython 5.4 and 6.1 *display()* is automatically made available to the user without import. If you are using display in a document that might be used in a pure python context or with older version of IPython, use the following import at the top of your file:

```
from IPython.display import display
```

Classes

Markdown([data, url, filename, metadata])

AFL.automation.shared.widgetui.Markdown

class AFL.automation.shared.widgetui.**Markdown**(*data=None*, *url=None*, *filename=None*, *metadata=None*)

Bases: TextDisplayObject

__init__(*data=None*, *url=None*, *filename=None*, *metadata=None*)

Create a display object given raw data.

When this object is returned by an expression or passed to the display function, it will result in the data being displayed in the frontend. The MIME type of the data should match the subclasses used, so the Png subclass should be used for 'image/png' data. If the data is a URL, the data will first be downloaded and then displayed.

Parameters

- **data** (*unicode, str or bytes*) – The raw data or a URL or file to load the data from
- **url** (*unicode*) – A URL to download the data from.
- **filename** (*unicode*) – Path to a local file to load the data from.
- **metadata** (*dict*) – Dict of metadata associated to be the object when displayed

Methods

__init__([data, url, filename, metadata])	Create a display object given raw data.
reload()	Reload the raw data from file or URL.

Attributes

metadata

__init__(*data=None*, *url=None*, *filename=None*, *metadata=None*)

Create a display object given raw data.

When this object is returned by an expression or passed to the display function, it will result in the data being displayed in the frontend. The MIME type of the data should match the subclasses used, so the Png subclass should be used for 'image/png' data. If the data is a URL, the data will first be downloaded and then displayed.

Parameters

- **data** (*unicode, str or bytes*) – The raw data or a URL or file to load the data from
- **url** (*unicode*) – A URL to download the data from.
- **filename** (*unicode*) – Path to a local file to load the data from.
- **metadata** (*dict*) – Dict of metadata associated to be the object when displayed

metadata = None

reload()

Reload the raw data from file or URL.

AFL.automation.shared.widgetui.**client_construct_ui**(*self*, *return_ui=False*, *display_ui=True*)

CHAPTER

SEVEN

MODULE DOCUMENTATION

7.1 AFL.automation

Functions

test()	Run all tests using pytest.

7.1.1 AFL.automation.test

AFL.automation.**test**()

Run all tests using pytest.

AFL.automation.**test**()

Run all tests using pytest.

Modules

APIServer
EpicsADLiveProcess
instrument
loading
prepare
sample_env
shared

7.1.2 AFL.automation.EpicsADLiveProcess

Modules

AreaDetectorLive
Client
ReduceDaemon

AFL.automation.EpicsADLiveProcess.AreaDetectorLive

Classes

AreaDetectorLive([basepv, cam, filewriter, ...])

AFL.automation.EpicsADLiveProcess.AreaDetectorLive.AreaDetectorLive

class AFL.automation.EpicsADLiveProcess.AreaDetectorLive.**AreaDetectorLive**(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

Bases: object

__init__(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

Methods

__init__([basepv, cam, filewriter, image])	
cbfunc([pvname, value, char_value])	
queuehandler()	
status()	

__init__(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

cbfunc(*pvname=None*, *value=None*, *char_value=None*, ***kwargs*)

queuehandler()

status()

class AFL.automation.EpicsADLiveProcess.AreaDetectorLive.**AreaDetectorLive**(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

__init__(*basepv='PIL5:'*, *cam='cam1:'*, *filewriter='TIFF1:'*, *image='image1:'*)

cbfunc(*pvname=None*, *value=None*, *char_value=None*, ***kwargs*)

queuehandler()

status()

AFL.automation.EpicsADLiveProcess.Client

Classes

Client([ip, port])	Communicate with NistoRoboto server on OT-2

AFL.automation.EpicsADLiveProcess.Client.Client

class AFL.automation.EpicsADLiveProcess.Client.**Client**(*ip='10.42.0.30'*, *port='5000'*)

Bases: object

Communicate with NistoRoboto server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the NistoRoboto server

__init__(*ip='10.42.0.30'*, *port='5000'*)

Methods

__init__([ip, port])	
halt()	
home()	
load_instrument(name, mount, tip_rack_slots)	
load_labware(name, slot)	
logged_in()	
login(username)	
reset()	
set_queue_mode([debug_mode])	
transfer(source, dest, volume[, source_loc, ...])	Transfer fluid from one location to another

__init__(*ip='10.42.0.30'*, *port='5000'*)

logged_in()

login(*username*)

set_queue_mode(*debug_mode=True*)

transfer(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str*) – Source wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

reset()

home()

halt()

class AFL.automation.EpicsADLiveProcess.Client.**Client**(*ip='10.42.0.30'*, *port='5000'*)

Communicate with NistoRoboto server on OT-2

This class maps pipettor functions to HTTP REST requests that are sent to the NistoRoboto server

__init__(*ip='10.42.0.30'*, *port='5000'*)

logged_in()

login(*username*)

set_queue_mode(*debug_mode=True*)

transfer(*source*, *dest*, *volume*, *source_loc=None*, *dest_loc=None*)

Transfer fluid from one location to another

Parameters

- **source** (*str or list of str*) – Source wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **dest** (*str or list of str*) – Destination wells to transfer from. Wells should be specified as three character strings with the first character being the slot number.
- **volume** (*float*) – volume of fluid to transfer in microliters

load_labware(*name*, *slot*)

load_instrument(*name*, *mount*, *tip_rack_slots*)

reset()

home()

halt()

AFL.automation.EpicsADLiveProcess.ReduceDaemon

Classes

ReduceDaemon(app, reduction_queue, ...[, ...])	

AFL.automation.EpicsADLiveProcess.ReduceDaemon.ReduceDaemon

class AFL.automation.EpicsADLiveProcess.ReduceDaemon.**ReduceDaemon**(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

Bases: Thread

__init__(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods

__init__(app, reduction_queue, integrator, ...)	This constructor should always be called with keyword arguments.
getName()	Return a string used for identification purposes only.
isDaemon()	Return whether this thread is a daemon.
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
run()	Method representing the thread's activity.
setDaemon(daemonic)	Set whether this thread is a daemon.
setName(name)	Set the name string for this thread.
start()	Start the thread's activity.
terminate()	

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread.
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
native_id	Native integral thread ID of this thread, or None if it has not been started.

__init__(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

run()

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

setDaemon(*daemonic*)

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

setName(*name*)

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class AFL.automation.EpicsADLiveProcess.ReduceDaemon.**ReduceDaemon**(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

__init__(*app*, *reduction_queue*, *integrator*, *results*, *mask=None*, *npts=500*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

terminate()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

D

E

F

G

H

I

J

K

M

N

O

P

Q

R

S

T

U

V

W

X

Z