AFL-automation

NIST AFL Team

May 07, 2025

CONTENTS:

AFL-automation is a framework for instrument control. It powers the NIST AFL, but is more useful than that. 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

MODULES

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

1.1 API Server

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

AFL.automation.APIServer

1.1.1 AFL.automation.APIServer

Modules

Client
LoggerFilter
QueueDaemon

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)	
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)	

continues on next page

Table 4 – continued from previous page

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.

continues on next page

Table 5 – continued from previous page

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.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(key[, default])	Retrieves the corresponding layout by the string key.
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()	

continues on next page

Table 10 – continued from previous page

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(*key*, *default=None*)

Retrieves the corresponding layout by the string key.

When there isn't an exact match, all the existing keys in the layout map will be treated as a regex and map against the input key again. When there are multiple matches for the regex, an *ValueError* will be raised. Returns *None* if there isn't any match found.

Parameters
key – String key to query a layout.

Returns
Corresponding layout based on the query.

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.

1.2 Instrument

The Instrument module contains drivers and interfaces for various scientific instruments.

AFL.automation.instrument

1.2.1 AFL.automation.instrument

Modules

FileCamera
NetworkCamera

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.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()

1.3 Loading

The Loading module provides functionality for loading and handling samples.

AFL.automation.loading

1.3.1 AFL.automation.loading

Modules

CetoniMultiPosValve	
ChemyxSyringePump	This is largely duplicated from the reference code provided by Chemyx.
DigitalOutPressureController	
DoubleViciMultiposSelector	
DummyPump	
FlowSelector	
MultiChannelRelay	
NE1kSyringePump	
PressureController	
PressureControllerAsPump	
SainSmartRelay	
SampleCell	
Sensor	
SensorCallbackThread	
SensorPollingThread	
SerialDevice	

continues on next page

Table 21 – continued from previous page

SyringePump	
Tubing	
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)	

continues on next page

Table 27 – continued from previous page

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()	

continues on next page

Table 28 – continued from previous page

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

class AFL.automation.loading.DoubleViciMultiposSelector.**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.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()	

continues on next page

Table 41 – continued from previous page

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.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()	

continues on next page

Table 49 – continued from previous page

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.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.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':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)

“””

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, ...])	

continues on next page

Table 69 – continued from previous page

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()	

continues on next page

Table 73 – continued from previous page

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.

continues on next page

Table 80 – continued from previous page

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.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

1.4 Prepare

The Prepare module provides functionality for sample preparation and processing.

1.5 Sample

The Sample module provides functionality for experiment orchestration 'Sample Servers'

AFL.automation.sample
AFL.automation.sample_env

1.5.1 AFL.automation.sample

Modules

1.5.2 AFL.automation.sample_env

Modules

1.6 Shared

The Shared module provides common utilities and functionality used across the AFL-automation framework.

AFL.automation.shared

1.6.1 AFL.automation.shared

Modules

DataLabelerWidget
DatasetWidget
DiffractionLabeler
MutableQueue
PersistentConfig
ServerDiscovery
exceptions
serialization
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*, */ (Positional-only parameter separator (PEP 570))*)

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()	

continues on next page

Table 107 – continued from previous page

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(** (Keyword-only parameters separator (PEP 3102))*, *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])

continues on next page

Table 114 – continued from previous page

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(iterable[, 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.

continues on next page

Table 115 – continued from previous page

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__().

classmethod fromkeys(*iterable*, *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(key[, default])	Retrieves the corresponding layout by the string key.
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(*key*, *default=None*)

Retrieves the corresponding layout by the string key.

When there isn't an exact match, all the existing keys in the layout map will be treated as a regex and map against the input key again. When there are multiple matches for the regex, an *ValueError* will be raised. Returns *None* if there isn't any match found.

Parameters
key – String key to query a layout.

Returns
Corresponding layout based on the query.

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()	

continues on next page

Table 129 – continued from previous page

get(key[, default])	Retrieves the corresponding layout by the string key.
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(*key*, *default=None*)

Retrieves the corresponding layout by the string key.

When there isn't an exact match, all the existing keys in the layout map will be treated as a regex and map against the input key again. When there are multiple matches for the regex, an *ValueError* will be raised. Returns *None* if there isn't any match found.

Parameters
key – String key to query a layout.

Returns
Corresponding layout based on the query.

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.

continues on next page

Table 130 – continued from previous page

IPVersion(value)	
RunThread(coro)	
ServerDiscovery()	ServerDiscovery class
ServiceBrowser(zc, type_[, handlers, ...])	Used to browse for a service of a specific type.
ServiceInfo	Service information.
ServiceStateChange(value)	
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

types
zc
query_scheduler
done
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.

types

zc

query_scheduler

done

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.

continues on next page

Table 133 – continued from previous page

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

text	
type	
key	
port	
weight	
priority	
server	
server_key	
host_ttl	
other_ttl	
interface_index	
addresses	IPv4 addresses of this service.
decoded_properties	Return properties as strings.

continues on next page

Table 134 – continued from previous page

name	The name of the service.
properties	Return properties as bytes.

text

type

key

port

weight

priority

server

server_key

host_ttl

other_ttl

interface_index

__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*)

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)	

continues on next page

Table 140 – continued from previous page

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

types	
zc	
query_scheduler	
done	
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	

types

zc

query_scheduler

done

__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

text
type

continues on next page

Table 144 – continued from previous page

key	
port	
weight	
priority	
server	
server_key	
host_ttl	
other_ttl	
interface_index	
addresses	IPv4 addresses of this service.
decoded_properties	Return properties as strings.
name	The name of the service.
properties	Return properties as bytes.

text

type

key

port

weight

priority

server

server_key

host_ttl

other_ttl

interface_index

__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*)

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.
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()	Wait for start up for actions that require a running Zeroconf instance.

continues on next page

Table 146 – continued from previous page

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() → 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.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

TWO

API REFERENCE

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

2.1 AFL.automation

Functions

test()	Run all tests using pytest.

AFL.automation.**test**()

Run all tests using pytest.

Modules

continues on next page

Table 3 – continued from previous page

sample_env
shared

2.1.1 AFL.automation.APIServer

Modules

Client
LoggerFilter
QueueDaemon

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.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.

2.1.2 AFL.automation.EpicsADLiveProcess

Modules

Client
ReduceDaemon

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.

2.1.3 AFL.automation.instrument

Modules

FileCamera
NetworkCamera

AFL.automation.instrument.FileCamera

Classes

FileCamera()

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

__init__()

collect(*fname*)

AFL.automation.instrument.NetworkCamera

Classes

NetworkCamera(url)

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

__init__(*url*)

camera_reset()

collect()

2.1.4 AFL.automation.loading

Modules

CetoniMultiPosValve	
ChemyxSyringePump	This is largely duplicated from the reference code provided by Chemyx.
DigitalOutPressureController	
DoubleViciMultiposSelector	
DummyPump	
FlowSelector	
MultiChannelRelay	
NE1kSyringePump	
PressureController	
PressureControllerAsPump	
SainSmartRelay	
SampleCell	
Sensor	
SensorCallbackThread	
SensorPollingThread	
SerialDevice	
SyringePump	
Tubing	
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()	
SerialDevice(port[, baudrate, timeout, ...])	
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.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.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.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.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

2.1.5 AFL.automation.sample_env

Modules

2.1.6 AFL.automation.shared

Modules

DataLabelerWidget
DatasetWidget
DiffractionLabeler
MutableQueue
PersistentConfig
ServerDiscovery
exceptions
serialization
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 date-time_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)	
RunThread(coro)	
ServerDiscovery()	ServerDiscovery class
ServiceBrowser(zc, type_[, handlers, ...])	Used to browse for a service of a specific type.
ServiceInfo	Service information.
ServiceStateChange(value)	
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

continues on next page

Table 50 – continued from previous page

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.widgetui

Functions

clear_output([wait])	Clear the output of the current cell receiving output.
client_construct_ui(self[, return_ui, ...])	

continues on next page

Table 53 – continued from previous page

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*)

2.2 AFL.automation.sample

Modules

CHAPTER

THREE

MODULE DOCUMENTATION

3.1 AFL.automation

Functions

test()	Run all tests using pytest.

3.1.1 AFL.automation.test

AFL.automation.**test**()

Run all tests using pytest.

AFL.automation.**test**()

Run all tests using pytest.

Modules

continues on next page

Table 3 – continued from previous page

instrument
loading
sample_env
shared

3.1.2 AFL.automation.EpicsADLiveProcess

Modules

Client
ReduceDaemon

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()

continues on next page

Table 6 – continued from previous page

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.

continues on next page

Table 8 – continued from previous page

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

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

H

I

J

K

L

M

N

O

P

S

T

U

V

W

Z