API reference for labbench

Release 0.20

Dan Kuester, Shane Allman, Paul Blanchard, Yao Ma (NIST)

Contents

1	Submodules	1
2	labbench.backends module	3
3	labbench.core module	27
4	labbench.data module	35
5	labbench.host module	47
6	labbench.notebooks module	55
7	labbench.util module	57
8	labbench.version module	67
9	Module contents	69
Рy	thon Module Index	71

CHAPTER 1

Submodules

labbench.backends module

class labbench.backends.CommandLineWrapper(resource=None, **settings)
Bases: labbench.core.Device

Virtual device representing for interacting with a command line executable. It supports threaded data logging through standard input, standard output, and standard error pipes.

On connection, the *backend* attribute is None. On a call to execute(), *backend* becomes is a subprocess instance. When EOF is reached on the executable's stdout, the backend is assumed terminated and is reset to None.

When *execute* is called, the program runs in a subprocess. The output piped to the command line standard output is queued in a background thread. Call read_stdout() to retreive (and clear) this queued stdout.

Parameters

- **arguments** (List ()) list of command line arguments to pass into the executable
- binary_path (Unicode ()) path to the file to run
- resource (Unicode ()) the data needed to make a connection to a device. its type and format are determined by the subclass implementation
- timeout (Float ()) Timeout (sec) after disconnect is called before killing the process

backend = DisconnectedBackend()

background(*extra_arguments, **flags)

Run the executable in the background (returning immediately while the executable continues running).

Once the background process is running, * Retreive standard output from the executable with self.read_stdout * Write to standard input self.write_stdin * Kill the process with self.kill * Check whether the process is running with self.running

Normally, the command line arguments are determined by * appending extra_arguments to the global arguments in self.settings.arguments, and * appending pairs of [key,value] from the *flags* dictionary to the

global flags defined with command flags in local state traits in self.settings

Use the self.no_state_arguments context manager to skip these global arguments like this:

```
with self.no_state_arguments:
    self.background(...)
```

Returns None

clear()

Clear queued standard output, discarding any contents

connect()

The *connect()* method exists to comply with the Device object protocol. Call the execute() method when connected to execute the binary.

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

exception_on_stderr

Use this context manager to raise exceptions if a process outputs to standard error during background execution.

```
foreground(*extra_arguments, **flags)
```

Blocking execution of the binary at the file location *self.settings.binary_path*.

Normally, the command line arguments are determined by * appending extra_arguments to the global arguments in self.settings.arguments, and * appending pairs of [key,value] from the *flags* dictionary to the

global flags defined with command flags in local state traits in self.settings

Use the self.no_state_arguments context manager to skip these global arguments like this:

```
with self.no_state_arguments:
    self.foreground(...)
```

Returns the return code of the process after its completion

kill()

no state arguments

Use this context manager to disable automatic use of state traits in generating argument strings.

```
read_stdout (wait_for=0)
```

Return string output queued from stdout for a process running in the background. This clears the queue.

Returns an empty string if the command line program has not been executed or is empty. Running the command line multiple times overwrites the queue.

Returns stdout

respawn

Use this context manager to respawning background execution.

running()

Return whether the executable is currently running

Returns True if running, otherwise False

```
class settings(device, *args, **kws)
```

Bases: labbench.core.settings

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• arguments: List

• arguments_min: Int

• binary_path: Unicode

• concurrency_support: Bool

• resource: Unicode

· timeout: Float

arguments

List()

list of command line arguments to pass into the executable

arguments_min

Int()

minimum number of extra command line arguments to pass to the executable

binary_path

Unicode()

path to the file to run

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

timeout

Float()

Timeout (sec) after disconnect is called before killing the process

```
class state(device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
write_stdin(text)
```

class labbench.backends.**DotNetDevice**(resource=None, **settings)

Bases: labbench.core.Device

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

backend = DisconnectedBackend()

connect()

DotNetDevice loads the DLL; importing in the class definition tries to load a lot of DLLs on import of ssmdevices, which would 1) break platforms that do not support the DLL, and 2) waste memory, and 3)

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
dll_name = None
```

library = None

```
class settings(device, *args, **kws)
```

Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- concurrency_support: Bool
- resource: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define (**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
class state (device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
class labbench.backends.EmulatedVISADevice (resource=None, **settings)
```

Bases: labbench.core.Device

Act as a VISA device without dispatching any visa commands

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
backend = DisconnectedBackend()
```

connect()

Backend implementations overload this to open a backend connection to the resource.

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
generators = {<class 'labbench.core.Bool'>: <function EmulatedVISADevice.<lambda>>, <</pre>
```

classmethod set_backend(backend_name)

backend_name can be 'py' or 'ni'

```
class settings(device, *args, **kws)
```

Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• concurrency_support: Bool

• resource: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

class state(device, *args, **kws)

Bases: labbench.backends.state

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

connected: Bool
identity: Unicode
options: Unicode
status_byte: Unicode

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

```
identity
```

Unicode()

options

Unicode()

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

status_byte

Unicode()

class labbench.backends.LabviewSocketInterface (resource=None, **settings)

Bases: labbench.core.Device

Implement the basic sockets-based control interface for labview. This implementation uses a transmit and receive socket.

State sets are implemented by simple 'command value' strings and implemented with the 'command' keyword (like VISA strings). Subclasses can therefore implement support for commands in specific labview VI the same was as in VISA commands by assigning the commands implemented in the corresponding labview VI.

The resource is the ip address of the host where the labview script is running. Use the tx_port and rx_port attributes to set the TCP/IP ports where communication is to take place.

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

backend = DisconnectedBackend()

clear()

Clear any data present in the read socket buffer.

connect()

Backend implementations overload this to open a backend connection to the resource.

delay = 1

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
read (convert_func=None)
```

Receive from the rx socket until self.rx_buffer_size samples are received or timeout happens after self.timeout seconds.

Optionally, apply the conversion function to the value after it is received.

```
resource = '127.0.0.1'
rx_buffer_size = 1024
rx_port = 61552
class settings(device, *args, **kws)
    Bases: labbench.core.settings
```

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- concurrency_support: Bool
- resource: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

```
classmethod define(**kws)
```

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
class state(device, *args, **kws)
Bases: labbench.core.state
```

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

```
classmethod setter(func)
```

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

A general base class for communication with serial devices. Unlike (for example) VISA instruments, there is no standardized command format like SCPI. The implementation is therefore limited to connect and disconnect, which open or close a pyserial connection object: the *link* attribute. Subclasses can read or write with the link attribute like they would any other serial instance.

A SerialDevice resource string is the same as the platform-dependent *port* argument to new serial. Serial objects.

Subclassed devices that need state descriptors will need to implement state_get and state_set methods in order to define how the state descriptors set and get operations.

Parameters

```
• dsrdtr(Bool()) -
• parity(Bytes()) -
• resource(Unicode()) - the data needed to make a connection to a device. its type and format are determined by the subclass implementation
• rtscts(Bool()) -
• stopbits(Float()) -
• timeout(Float()) -
• write_termination(Bytes()) -
• xonxoff(Bool()) -

backend = DisconnectedBackend()
connect()
Connect to the serial device with the VISA resource string defined in self.settings.resource
```

• baud rate(Int())-

disconnect()

Disconnect the serial instrument

classmethod from_hwid(hwid=None, *args, **connection_params)

Instantiate a new SerialDevice from a 'hwid' resource instead of a comport resource. A hwid string in windows might look something like:

r'PCIVEN 8086&DEV 9D3D&SUBSYS 06DC1028&REV 213&11583659&1&B3'

static list ports(hwid=None)

List USB serial devices on the computer

Returns list of port resource information

```
class settings(device, *args, **kws)
```

Bases: labbench.core.settings

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- baud_rate: Int
- concurrency_support: Bool
- dsrdtr: Bool
- · parity: Bytes
- resource: Unicode
- rtscts: Bool
- stopbits: Float
- timeout: Float
- write termination: Bytes
- xonxoff: Bool

baud_rate

Int()

concurrency_support

Bool()

whether this $\ensuremath{\operatorname{Device}}$ implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default value=7*.

dsrdtr

Bool()

parity

Bytes()

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

rtscts

Bool()

stopbits

Float()

timeout

Float()

write_termination

Bytes()

xonxoff

Bool()

class state(device, *args, **kws)

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

windows might look something like:

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in self._device. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
class labbench.backends.SerialLoggingDevice(resource=None, **settings)
    Bases: labbench.backends.SerialDevice
```

Manage connection, acquisition, and data retreival on a single GPS device. The goal is to make GPS devices controllable somewhat like instruments; maintaining their own threads, and blocking during setup or stop command execution.

Listener objects must implement an attach method with one argument consisting of the queue that the device manager uses to push data from the serial port.

```
Parameters
          • baud rate(Int())-
          • data_format (Bytes ()) - Data format metadata
          • dsrdtr(Bool())-
          • max queue size (Int ()) - Number of bytes to allocate in the data retreival buffer
          • parity (Bytes ()) -
          • poll_rate (Float ()) - Data retreival rate from the device (in seconds)
          • resource (Unicode ()) – the data needed to make a connection to a device. its type and
            format are determined by the subclass implementation
          • rtscts(Bool())-
          • stop_timeout (Float ()) - Delay after a call to stop before terminating the runloop
            thread
          • stopbits (Float ()) -
          • timeout (Float ()) -
          • write_termination(Bytes())-
          • xonxoff(Bool())-
backend = DisconnectedBackend()
clear()
configure()
    Connect to the serial device with the VISA resource string defined in self.settings.resource
disconnect()
    Disconnect the serial instrument
fetch()
classmethod from_hwid(hwid=None, *args, **connection_params)
    Instantiate a new SerialDevice from a 'hwid' resource instead of a comport resource. A hwid string in
```

r'PCIVEN_8086&DEV_9D3D&SUBSYS_06DC1028&REV_213&11583659&1&B3'

```
static list_ports(hwid=None)
```

List USB serial devices on the computer

Returns list of port resource information

running()

```
class settings(device, *args, **kws)
```

Bases: labbench.backends.settings

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- baud_rate: Int
- concurrency_support: Bool
- data_format: Bytes
- · dsrdtr: Bool
- max_queue_size: Int
- · parity: Bytes
- poll_rate: Float
- resource: Unicode
- · rtscts: Bool
- stop_timeout: Float
- stopbits: Float
- timeout: Float
- write_termination: Bytes
- xonxoff: Bool

baud_rate

Int()

concurrency_support

Bool()

whether this Device implementation supports threading

data_format

Bytes()

Data format metadata

```
classmethod define(**kws)
```

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

dsrdtr

Bool()

max_queue_size

Int()

Number of bytes to allocate in the data retreival buffer

parity

Bytes()

poll_rate

Float()

Data retreival rate from the device (in seconds)

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

rtscts

Bool()

stop_timeout

Float()

Delay after a call to stop before terminating the runloop thread

stopbits

Float()

timeout

Float()

write_termination

Bytes()

xonxoff

Bool()

start()

Start a background thread that acquires log data into a queue.

```
class state (device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
stop()
```

```
class labbench.backends.TelnetDevice(resource=None, **settings)
    Bases: labbench.core.Device
```

A general base class for communication devices via telnet. Unlike (for example) VISA instruments, there is no standardized command format like SCPI. The implementation is therefore limited to connect and disconnect, which open or close a pyserial connection object: the *backend* attribute. Subclasses can read or write with the backend attribute like they would any other telnetlib instance.

A SerialDevice resource string is the same as the platform-dependent *port* argument to new serial. Serial objects.

Subclassed devices that need state descriptors will need to implement state_get and state_set methods in order to define how the state descriptors set and get operations.

Parameters

- **port** (Int ()) -
- **resource** (Unicode ()) the data needed to make a connection to a device. its type and format are determined by the subclass implementation
- timeout (Float ()) -

backend = DisconnectedBackend()

connect()

Make the telnet connection to the host defined by the string in self.settings.resource

disconnect()

Disconnect the telnet connection

```
class settings(device, *args, **kws)
```

Bases: labbench.core.settings

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• concurrency_support: Bool

• port: Int

• resource: Unicode

· timeout: Float

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define (**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

port

Int()

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

timeout

Float()

```
class state (device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
class labbench.backends.VISADevice(resource=None, **settings)
```

```
Bases: labbench.core.Device
```

```
class VISADevice (resource, read_termination='\n', write_termination='\n')
```

VISADevice instances control VISA instruments using a pyvisa backend. Compared to direct use of pyvisa, this style of use permits use of labbench device *state* goodies for compact, readable code, as well as type checking.

For example, the following fetches the identity string from the remote instrument:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as instr:
    print inst.state.identity
```

This is equivalent to the more pyvisa-style use as follows:

```
inst = VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR')
inst.connect()
print inst.query('*IDN?')
```

Use of *inst.state* makes it possible to add callbacks to support automatic state logging, or to build a UI.

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
backend = DisconnectedBackend()
```

connect()

Connect to the VISA instrument defined by the VISA resource set by *self.settings.resource*. The pyvisa backend object is assigned to *self.backend*.

Returns None

Instead of calling *connect* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as inst:
    print inst.state.identity
    print inst.state.status_byte
    print inst.state.options
```

would instantiate a *VISADevice* and guarantee it is disconnected either at the successful completion of the *with* block, or if there is any exception.

disconnect()

Disconnect the VISA instrument. If you use a *with* block this is handled automatically and you do not need to call this method.

Returns None

```
classmethod list_resources()
```

```
overlap_and_block (timeout=None, quiet=False)
```

Append ';*OPC' to each command sent during the block, then send an '*OPC?' query to block until the measurement is done

This is meant to be used in *with* blocks as follows:

```
with inst.overlap_and_block():
   inst.write('long running command 1')
   inst.write('long running command 2')
```

At the end of the with block, a '*OPC?' query is sent that blocks until the instrument signals it has finished with the commands.

Optionally, VISA timeout exceptions can be suppressed with the *quiet* argument.

Parameters

- timeout Timeout, in milliseconds, or None to use self.backend.timeout
- quiet Suppress timeout exceptions if this evaluates as True

preset()

Convenience function to send standard SCPI '*RST'

```
query (msg, timeout=None)
```

Query an SCPI command to the device with pyvisa, and return a string containing the device response.

Handles debug logging, and extra *OPC commands (when in an overlap_and_block context), then sends a command string to the device.

Parameters msg(str) – the SCPI command to send by VISA

Returns the response to the query from the device

classmethod set_backend(backend_name)

Set the pyvisa resource manager for all VISA objects.

```
Parameters str (backend_name) - '@ni' (the default) or '@py'
```

Returns None

```
class settings(device, *args, **kws)
Bases: labbench.core.settings
```

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• concurrency_support: Bool

• read_termination: Unicode

• resource: Unicode

• write termination: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

read_termination

Unicode()

termination character to indicate end of message on receive from the instrument

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

write_termination

Unicode()

termination character to indicate end of message in messages sent to the instrument

```
class state (device, *args, **kws)
Bases: labbench.core.state
```

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

connected: Bool
identity: Unicode
options: Unicode
status_byte: Dict

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self_device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

identity

Unicode()

identity string reported by the instrument

options

Unicode()

options reported by the instrument

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

status_byte

Dict()

VISA status byte reported by the instrument

class suppress_timeout (*exceptions)

Bases: contextlib.suppress

Context manager to suppress timeout exceptions.

Example:

```
with inst.suppress_timeout():
   inst.write('long running command 1')
   inst.write('long running command 2')
```

If the command 1 raises an exception, then command 2 will (silently) not execute.

wait()

Convenience function to send standard SCPI '*WAI'

write (msg)

Write an SCPI command to the device with pyvisa.

Handles debug logging, and extra *OPC commands (when in an overlap_and_block context), then sends a command string to the device.

Parameters msg (str) - the SCPI command to send by VISA

Returns None

```
class labbench.backends.Win32ComDevice(resource=None, **settings)
Bases: labbench.core.Device
```

Basic support for calling win32 COM APIs.

The python wrappers for COM drivers still basically require that threading is performed using the windows COM API, and not the python threading. Figuring this out with win32com calls within python is not for the faint of heart. Threading support is instead realized with util.ThreadSandbox, which ensures that all calls to the dispatched COM object block until the previous calls are completed from within a background thread. Set concurrency_support=True to decide whether this thread support wrapper is applied to the dispatched Win32Com object.

Parameters

- com_object (Unicode ()) -
- **concurrency_support** (Bool ()) whether this Device implementation supports threading
- resource (Unicode ()) the data needed to make a connection to a device. its type and format are determined by the subclass implementation

backend = DisconnectedBackend()

connect()

Connect to the win32 com object

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
class settings(device, *args, **kws)
Bases: labbench.core.settings
```

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• com_object: Unicode

• concurrency_support: Bool

• resource: Unicode

com_object

Unicode()

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
class state(device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self. device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

CHAPTER 3

labbench.core module

This implementation is deeply intertwined with low-level internals of traitlets and obscure details of the python object model. Consider reading the documentation closely and inheriting these objects instead of reverse-engineering this code.

```
exception labbench.core.ConnectionError
     Bases: traitlets.traitlets.TraitError
     args
     with_traceback()
         Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.
exception labbench.core.DeviceNotReady
     Bases: Exception
     args
     with_traceback()
         Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.
exception labbench.core.DeviceFatalError
     Bases: Exception
     args
     with traceback()
         Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.
exception labbench.core.DeviceException
     Bases: Exception
     args
     with_traceback()
         Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.
exception labbench.core.DeviceConnectionLost
     Bases: Exception
```

```
args
    with_traceback()
         Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.
exception labbench.core.DeviceStateError
    Bases: traitlets.traitlets.TraitError
    args
    with_traceback()
         Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
class labbench.core.Int (default_value=traitlets.Undefined, allow_none=False, read_only=None,
                            help=None, write only=None, cache=None, command=None, get-
                            ter=None, setter=None, remap={}, **kwargs)
    Bases: labbench.core.TraitMixIn, traitlets.traitlets.CInt
    doc_attrs = ('min', 'max', 'command', 'read_only', 'write_only', 'remap', 'cache')
     info_text = 'an int'
    validate(obj, value)
class labbench.core.Float (default value=traitlets.Undefined,
                                                                        allow none=False,
                              read only=None, help=None, write only=None, cache=None, com-
                              mand=None, getter=None, setter=None, remap={}, **kwargs)
    Bases: labbench.core.TraitMixIn, labbench.core.CFLoatSteppedTraitlet
    doc_attrs = ('min', 'max', 'step', 'command', 'read_only', 'write_only', 'remap', 'cac'
     info_text = 'a float'
    validate(obj, value)
class labbench.core.Unicode (default value=traitlets.Undefined,
                                                                        allow none=False.
                                 read_only=None, help=None, write_only=None, cache=None,
                                 command=None, getter=None, setter=None, remap={}, **kwargs)
    Bases: labbench.core.TraitMixIn, traitlets.traitlets.CUnicode
    default_value = ''
    info_text = 'a unicode string'
    validate(obj, value)
class labbench.core.Complex (default_value=traitlets.Undefined,
                                                                        allow none=False,
                                 read_only=None, help=None, write_only=None, cache=None,
                                 command=None, getter=None, setter=None, remap={}, **kwargs)
    Bases: labbench.core.TraitMixIn, traitlets.traitlets.CComplex
    info_text = 'a complex number'
    validate(obj, value)
class labbench.core.Bytes (default_value=traitlets.Undefined,
                                                                        allow none=False,
                              read_only=None, help=None, write_only=None, cache=None, com-
                              mand=None, getter=None, setter=None, remap={}, **kwargs)
    Bases: labbench.core.TraitMixIn, traitlets.traitlets.CBytes
    info_text = 'a bytes object'
    validate(obj, value)
```

```
class labbench.core.CaselessBytesEnum(default_value=traitlets.Undefined,
                                                                                         al-
                                              low_none=False.
                                                               read only=None,
                                                                                 help=None,
                                              write only=None, cache=None, command=None,
                                              getter=None, setter=None, remap={}, **kwargs)
     Bases: labbench.core.TraitMixIn, labbench.core.EnumBytesTraitlet
     devalidate(obj, value)
     doc_attrs = ('values', 'case_sensitive', 'command', 'read_only', 'write_only', 'remap'
     info_text = 'a bytes object'
     validate(obj, value)
class labbench.core.Bool(default_value=traitlets.Undefined, allow_none=False, read_only=None,
                              help=None, write_only=None, cache=None, command=None, get-
                              ter=None, setter=None, remap={}, **kwargs)
     Bases: labbench.core.TraitMixIn, traitlets.traitlets.CBool
     default value = False
     info_text = 'a boolean'
     validate(obj, value)
class labbench.core.List (default_value=traitlets.Undefined, allow_none=False, read_only=None,
                              help=None, write_only=None, cache=None, command=None, get-
                              ter=None, setter=None, remap={}, **kwargs)
     Bases: labbench.core.TraitMixIn, traitlets.traitlets.List
     class init (cls. name)
         Part of the initialization which may depend on the underlying HasDescriptors class.
         It is typically overloaded for specific types.
         This method is called by MetaHasDescriptors.__init__() passing the class (cls) and name un-
         der which the descriptor has been assigned.
     default_value = []
     default_value_repr()
     element_error (obj, element, validator)
     error (obj, value)
     info()
     instance init(obj)
         Part of the initialization which may depend on the underlying HasDescriptors instance.
         It is typically overloaded for specific types.
         This method is called by HasTraits.__new__() and in the BaseDescriptor.
         instance_init() method of descriptors holding other descriptors.
     klass
         alias of builtins.list
     length_error (obj, value)
     make_dynamic_default()
     validate(obj, value)
     validate_elements(obj, value)
```

```
class labbench.core.Dict (default value=traitlets.Undefined, allow none=False, read only=None,
                               help=None, write_only=None, cache=None, command=None, get-
                               ter=None, setter=None, remap={}, **kwargs)
     Bases: labbench.core.TraitMixIn, traitlets.traitlets.Dict
     class_init (cls, name)
         Part of the initialization which may depend on the underlying HasDescriptors class.
         It is typically overloaded for specific types.
         This method is called by MetaHasDescriptors.__init__() passing the class (cls) and name un-
         der which the descriptor has been assigned.
     default_value_repr()
     element_error (obj, element, validator)
     error (obj, value)
     info()
     instance init(obj)
         Part of the initialization which may depend on the underlying HasDescriptors instance.
         It is typically overloaded for specific types.
         This method is called by HasTraits.__new__() and in the BaseDescriptor.
         instance_init () method of descriptors holding other descriptors.
     klass = None
     make_dynamic_default()
     validate(obj, value)
     validate_elements(obj, value)
class labbench.core.TCPAddress (default_value=traitlets.Undefined,
                                                                            allow_none=False,
                                       read_only=None, help=None, write_only=None, cache=None,
                                       command=None, getter=None, setter=None,
                                       **kwargs)
     Bases: labbench.core.TraitMixIn, traitlets.traitlets.TCPAddress
     info_text = 'an (ip, port) tuple'
     validate(obj, value)
class labbench.core.CaselessStrEnum(default_value=traitlets.Undefined,
                                                                            allow_none=False,
                                             read_only=None,
                                                               help=None,
                                                                             write_only=None,
                                             cache=None, command=None, getter=None, set-
                                            ter=None, remap={}, **kwargs)
     Bases: labbench.core.TraitMixIn, traitlets.traitlets.CaselessStrEnum
     doc_attrs = ('values', 'command', 'read_only', 'write_only', 'remap', 'cache')
         Returns a description of the trait.
     validate(obj, value)
class labbench.core.Device(resource=None, **settings)
     Bases: object
     Device is the base class common to all labbench drivers. Inherit it to implement a backend, or a specialized type
```

of driver.

Drivers that subclass Device get

- device connection management via context management (the *with* statement)
- test state management for easy test logging and extension to UI
- · a degree automatic stylistic consistency between drivers

Parameters

- **resource** resource identifier, with type and format determined by backend (see specific subclasses for details)
- **local_states set the local state for each supplied state key and value

Note: Use Device by subclassing it only if you are implementing a driver that needs a new type of backend.

Several types of backends have already been implemented as part of labbench:

- VISADevice exposes a pyvisa backend for VISA Instruments
- CommandLineWrapper exposes a threaded pipes backend for command line tools
- Serial exposes a pyserial backend for serial port communication
- DotNetDevice exposes a pythonnet for wrapping dotnet libraries

(and others). If you are implementing a driver that uses one of these backends, inherit from the corresponding class above, not *Device*.

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

backend = DisconnectedBackend()

it is to be set in *connect* and *disconnect* by the subclass that implements the backend.

connect()

Backend implementations overload this to open a backend connection to the resource.

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
class settings(device, *args, **kws)
```

Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

• concurrency_support: Bool

• resource: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

class state(device, *args, **kws)

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

labbench.core.list_devices(depth=1)

Look for Device instances, and their names, in the calling code context (depth == 1) or its callers (if depth in (2,3,...)). Checks locals() in that context first. If no Device instances are found there, search the first argument of the first function argument, in case this is a method in a class.

exception labbench.core.CommandNotImplementedError

Bases: NotImplementedError

args

with_traceback()

Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.

CHAPTER 4

labbench.data module

class labbench.data.StateAggregator

Bases: object

Aggregate state information from multiple devices. This can be the basis for automatic database logging.

get()

Aggregate and return the current device states as configured with observe ().

Returns dictionary of aggregated states. Keys are str formatted

according to the key() (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

key (*device_name*, *state_name*)

Generate a name for a state based on the names of a device and one of its remotelets

observe (devices, changes=True, always=[], never=[])

observe_settings (devices, changes=True, always=[], never=['connected'])

Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to observe_settings() replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- always name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** name (or iterable of multiple names) of settings to exclude from aggregated result (overrides :param:'always')

```
observe states (devices, changes=True, always=[], never=['connected'])
```

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_states()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- changes (bool) Whether to automatically log each time a state is set for the supplied device(s)
- **always** name (or iterable of multiple names) of states to actively update on each call to get()
- never name (or iterable of multiple names) of states to exclude from aggregated result (overrides :param:'always')

```
set_device_labels(**mapping)
```

Manually choose device name for a device instance.

Parameters mapping (dict) – name mapping, formatted as {device_object: 'device name'}

Returns None

```
class labbench.data.StatesToRelationalTable (path, overwrite = False, text\_relational\_min = 1024, force\_relational = ['host\_log'], dirname\_fmt = '\{id\} \{host\_time\}', nonscalar\_file\_type = 'csv', meta-data\_dirname = 'metadata', tar = False, **metadata')
```

 $Bases: \ \textit{labbench.data.StateAggregator}$

Abstract base class for loggers that queue dictionaries of data before writing to disk. This extends StateAggregator to support

- 1. queuing aggregate state of devices by lists of dictionaries;
- 2. custom metadata in each queued aggregate state entry; and
- 3. custom response to non-scalar data (such as relational databasing).

Parameters

- path (str) path to use for the master database
- **overwrite** (bool) whether to overwrite the master database if it exists (otherwise, append)

```
append (*args, **kwargs)
```

Add a new row of data to the list of data that awaits write to disk, *self.pending*. The row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe_states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with *self.observe_states*.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in set_path_format()), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until write().

Parameters copy=True (bool) - When True (the default), a deep copy o

data is used to avoid problems with overwriting references to data if data is reused during test. This takes some extra time; set to False to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()

Remove any queued data that has been added by append.

close(

Close the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

get()

Aggregate and return the current device states as configured with observe ().

Returns dictionary of aggregated states. Keys are str formatted

according to the key() (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

```
index label = 'id'
```

key (device name, state name)

Generate a name for a state based on the names of a device and one of its remotelets

observe (devices, changes=True, always=[], never=[])

```
observe_settings (devices, changes=True, always=[], never=['connected'])
```

Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_settings()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- **always** name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** name (or iterable of multiple names) of settings to exclude from aggregated result (overrides :param:'always')

observe states (devices, changes=True, always=[], never=['connected'])

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_states()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- always name (or iterable of multiple names) of states to actively update on each call to get()
- never name (or iterable of multiple names) of states to exclude from aggregated result (overrides:param:'always')

open (path=None)

set_device_labels(**mapping)

Manually choose device name for a device instance.

Parameters mapping (dict) – name mapping, formatted as {device_object: 'device name'}

Returns None

set path format(format)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

Parameters format – a string compatible with str.format(), with replacement

fields defined from the keys from the current entry of results and aggregated states.

The format string follows the syntax of python's python's built-in str.format(). You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes <code>inst1_frequency</code> of <code>915e6</code>, and <code>append()</code> has been called as <code>append(dut="DUT15")</code>. If the current aggregate state entry includes <code>inst1_frequency=915e6</code>, then the format string '{dut}/{inst1_frequency}' means relative data path 'DUT15/915e6'.

Returns None

$\verb|set_relational_file_format| (format)$

Set the format to use for relational data files.

Parameters format (str) - one of 'csv', 'json', 'feather', or 'pickle'

set_row_preprocessor (func)

Define a function that is called to modify each pending data row before it is committed to disk. It should accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()

Open the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

write()

Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

Add a new row of data to the list of data that awaits write to disk, *self.pending*. The row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with self.observe_states.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in set_path_format()), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until write().

```
Parameters copy=True (bool) - When True (the default), a deep copy o
```

data is used to avoid problems with overwriting references to data if data is reused during test. This takes some extra time; set to False to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()

Remove any queued data that has been added by append.

close()

Close the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

get()

Aggregate and return the current device states as configured with observe ().

Returns dictionary of aggregated states. Keys are str formatted

according to the key() (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

```
index_label = 'id'
```

```
key (device name, state name)
```

Generate a name for a state based on the names of a device and one of its remotelets

```
nonscalar_file_type = 'csv'
```

```
observe (devices, changes=True, always=[], never=[])
```

```
observe_settings (devices, changes=True, always=[], never=['connected'])
```

Configure Each time a device setting is set from python, intercept the value to include in the aggregate state

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to observe_settings() replace the existing list of observed states for each device.

Parameters

- devices Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- always name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** name (or iterable of multiple names) of settings to exclude from aggregated result (overrides :param:'always')

```
observe_states (devices, changes=True, always=[], never=['connected'])
```

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_states()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- **always** name (or iterable of multiple names) of states to actively update on each call to get()
- never name (or iterable of multiple names) of states to exclude from aggregated result (overrides :param:'always')

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with StatesToCSV('my.csv') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *StatesToCSV* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

```
set_device_labels(**mapping)
```

Manually choose device name for a device instance.

Parameters mapping (dict) – name mapping, formatted as {device_object: 'device name'}

Returns None

set_path_format (format)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

Parameters format - a string compatible with str.format(), with replacement

fields defined from the keys from the current entry of results and aggregated states.

The format string follows the syntax of python's python's built-in str.format(). You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes <code>instl_frequency</code> of <code>915e6</code>, and <code>append()</code> has been called as <code>append(dut="DUT15")</code>. If the current aggregate state entry includes <code>instl_frequency=915e6</code>, then the format string '{dut}/{instl_frequency}' means relative data path 'DUT15/915e6'.

Returns None

set_relational_file_format(format)

Set the format to use for relational data files.

```
Parameters format (str) - one of 'csv', 'json', 'feather', or 'pickle'
```

set_row_preprocessor (func)

Define a function that is called to modify each pending data row before it is committed to disk. It should accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()

Open the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

write()

Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

Add a new row of data to the list of data that awaits write to disk, *self.pending*. The row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe_states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with self.observe_states.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in set_path_format()), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until write().

Parameters copy=True (bool) - When True (the default), a deep copy o

data is used to avoid problems with overwriting references to data if data is reused during test. This takes some extra time; set to False to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()

Remove any queued data that has been added by append.

close()

Close the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

get()

Aggregate and return the current device states as configured with observe ().

Returns dictionary of aggregated states. Keys are str formatted

according to the key() (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

```
index_label = 'id'
```

key (name, attr)

The key determines the SQL column name. df.to_sql does not seem to support column names that include spaces

```
master_filename = 'master.db'
```

observe (devices, changes=True, always=[], never=[])

```
observe settings (devices, changes=True, always=[], never=['connected'])
```

Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_settings()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- always name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** name (or iterable of multiple names) of settings to exclude from aggregated result (overrides :param:'always')

observe states (devices, changes=True, always=[], never=['connected'])

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to <code>observe_states()</code> replace the existing list of observed states for each device.

Parameters

- **devices** Device instance or iterable of Device instances
- **changes** (bool) Whether to automatically log each time a state is set for the supplied device(s)
- always name (or iterable of multiple names) of states to actively update on each call to get()
- **never** name (or iterable of multiple names) of states to exclude from aggregated result (overrides :param:'always')

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with StatesToSQLite('my.db') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *StatesToCSV* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

set_device_labels(**mapping)

Manually choose device name for a device instance.

Parameters mapping (dict) – name mapping, formatted as {device_object: 'device name'}

Returns None

set_path_format (format)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

Parameters format - a string compatible with str.format(), with replacement

fields defined from the keys from the current entry of results and aggregated states.

The format string follows the syntax of python's python's built-in str.format(). You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes <code>inst1_frequency</code> of <code>915e6</code>, and <code>append()</code> has been called as <code>append(dut="DUT15")</code>. If the current aggregate state entry includes <code>inst1_frequency=915e6</code>, then the format string '{dut}/{inst1_frequency}' means relative data path 'DUT15/915e6'.

Returns None

set relational file format(format)

Set the format to use for relational data files.

Parameters format (str) - one of 'csv', 'json', 'feather', or 'pickle'

set_row_preprocessor (func)

Define a function that is called to modify each pending data row before it is committed to disk. It should

accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()

Open the file or database connection. This is an abstract base method (it be overridden by inheriting classes)

Returns None

```
table_name = 'master'
write()
```

Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

labbench.data.read (path_or_buf, columns=None, nrows=None, format='auto', **kws)
Read tabular data from a file in one of various formats using pandas.

Parameters

- path (str) path to the data file.
- **columns** a column or iterable of multiple columns to return from the data file, or None (the default) to return all columns
- nrows number of rows to read at the beginning of the table, or None (the default) to read all rows
- **format** (str) data file format, one of ['pickle','feather','csv','json','csv'], or 'auto' (the default) to guess from the file extension
- **kws** additional keyword arguments to pass to the pandas read_<ext> function matching the file extension

Returns pandas. DataFrame instance containing data read from file

```
labbench.data.read_relational(path, expand_col, master_cols=None, target_cols=None, master_nrows=None, master_format='auto', prepend_column_name=True)
```

Flatten a relational database table by loading the table located each row of *master[expand_col]*. The value of each column in this row is copied to the loaded table. The columns in the resulting table generated on each row are downselected according to *master_cols* and *target_cols*. Each of the resulting tables is concatenated and returned.

The expanded dataframe may be very large, making downselecting a practical necessity in some scenarios.

TODO: Support for a list of expand_col?

Parameters

- master (pandas. DataFrame) the master database, consisting of columns containing data and columns containing paths to data files
- **expand_col** (str) the column in the master database containing paths to data files that should be expanded
- master_cols a column (or array-like iterable of multiple columns) listing the master columns to include in the expanded dataframe, or None (the default) pass all columns from master
- target_cols a column (or array-like iterable of multiple columns) listing the master columns to include in the expanded dataframe, or None (the default) to pass all columns loaded from each master[expand_col]

- master_path a string containing the full path to the master database (to help find the relational files)
- **prepend_column_name** (bool) whether to prepend the name of the expanded column from the master database

Returns the expanded dataframe

labbench.data.to_feather(data, path)

Write a dataframe to a feather file on disk. Any index will be moved to a column, index and column name metadata will be removed, and columns names will be changed to a string.

Parameters

- data dataframe to write to disk
- path path to file to write

Returns None

CHAPTER 5

labbench.host module

```
class labbench.host.Host (resource=None, **settings)
     Bases: labbench.core.Device
         Parameters resource (Unicode ()) - the data needed to make a connection to a device. its type
             and format are determined by the subclass implementation
     backend = DisconnectedBackend()
     connect()
         The host setup method tries to commit current changes to the tree
     disconnect()
         Backend implementations must overload this to disconnect an existing connection to the resource encap-
         sulated in the object.
     get_git_browse_url
         Unicode()
     get_git_commit
         Unicode()
     get_git_remote_url
         Unicode()
     get_hostname
         Unicode()
     get_log
         Unicode()
     get_time
     log_format = '%(asctime)s.%(msecs).03d %(levelname)10s %(message)s'
     metadata()
     class settings(device, *args, **kws)
         Bases: labbench.core.HasSettingsTraits
```

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- concurrency_support: Bool
- resource: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
class state(device, *args, **kws)
    Bases: labbench.core.state
```

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- · connected: Bool
- git_browse_url: Unicode
- git commit id: Unicode
- git_remote_url: Unicode
- hostname: Unicode

```
• log: Unicode
```

• time: Unicode

connected Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

```
git_browse_url
     Unicode()

git_commit_id
     Unicode()

git_remote_url
     Unicode()

hostname
     Unicode()

log
     Unicode()
```

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

time

Unicode()

```
time_format = '%Y-%m-%d %H:%M:%S'
class labbench.host.Email(resource=None, **settings)
    Bases: labbench.core.Device
```

Sends a notification message on disconnection. If an exception was thrown, this is a failure subject line with traceback information in the main body. Otherwise, the message is a success message in the subject line. Stderr is also sent.

Parameters

- failure_message (Unicode ()) subject line for test failure emails, or None to suppress success emails
- recipients (List ()) -
- resource (TCPAddress())-

- sender (Unicode ()) -
- success_message (Unicode ()) subject line for test success emails, or None to suppress success emails

backend = DisconnectedBackend()

connect()

Backend implementations overload this to open a backend connection to the resource.

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
send_summary()
```

```
class settings(device, *args, **kws)
```

Bases: labbench.core.settings

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- concurrency_support: Bool
- failure_message: Unicode
- · recipients: List
- resource: TCPAddress
- sender: Unicode
- success_message: Unicode

concurrency_support

Bool()

whether this Device implementation supports threading

classmethod define(**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

failure message

Unicode()

subject line for test failure emails, or None to suppress success emails

recipients

List()

resource

TCPAddress()

sender

Unicode()

success_message

Unicode()

subject line for test success emails, or None to suppress success emails

class state (device, *args, **kws)

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self_device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

```
class labbench.host.LogStderr(resource=None, **settings)
    Bases: labbench.core.Device
```

This "Device" logs a copy of messages on sys.stderr while connected.

Parameters resource (Unicode ()) – the data needed to make a connection to a device. its type and format are determined by the subclass implementation

backend = DisconnectedBackend()

connect()

Backend implementations overload this to open a backend connection to the resource.

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

```
flush()
```

log = ''

```
class settings (device, *args, **kws)
```

Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device. These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- concurrency_support: Bool
- resource: Unicode

concurrency_support

Bool()

whether this $\ensuremath{\operatorname{Device}}$ implementation supports threading

classmethod define (**kws)

Return a copy of this class with default values of the traits redefined according to each keyword argument. For example:

```
mylocaltraitcls.define(parameter=7)
```

will return a deep copy of *mylocaltraitcls* where its trait named *parameter* is redefined to have *default_value=7*.

resource

Unicode()

the data needed to make a connection to a device. its type and format are determined by the subclass implementation

```
class state(device, *args, **kws)
```

Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assignment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

· connected: Bool

connected

Bool()

whether the Device instance is connected

classmethod getter(func)

Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(func)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

write(what)

CHAPTER 6

labbench.notebooks module

```
class labbench.notebooks.panel
```

Bases: object

Show tables summarizing device settings and states in jupyter notebook. Only a single panel will be shown in a python kernel.

Parameters source – Either an integer indicating how far up the calling tree to search

for Device instances, or a *labbench.Testbed* instance. :param ncols: Maximum number of devices to show on each row

```
children = []
devices = {}
ncols = 2
widget = None
```

labbench.notebooks.log_progress (sequence, every=None, size=None, title=None) Indicate slow progress through a long sequence.

This code is adapted here from https://github.com/alexanderkuk/log-progress where it was provided under the MIT license.

Parameters

- sequence iterable to monitor
- every the number of iterations to skip between updating the progress bar, or None to update all
- **size** number of elements in the sequence (required only for generators with no length estimate)
- title title text

Returns iterator that yields the elements of *sequence*

labbench.notebooks.range(*args, **kws)

the same as python range, but with a progress bar representing progress iterating through the range

labbench.notebooks.linspace(*args, **kws)

the same as numpy.linspace, but with a progress bar representing progress iterating through the range, and an optional title= keyword argument to set the title

CHAPTER 7

labbench.util module

```
labbench.util.concurrently(*objs, **kws)
```

If *objs are callable (like functions), call each of *objs in concurrent threads. If *objs are context managers (such as Device instances to be connected), enter each context in concurrent threads.

Multiple references to the same function in *funcs* only result in one call. The *catch* and *flatten* arguments may be callables, in which case they are executed (and each flag value is treated as defaults).

Parameters

- **objs** each argument may be a callable (function or class that defines a __call__ method), or context manager (such as a Device instance)
- **catch** if *False* (the default), a *ConcurrentException* is raised if any of *funcs* raise an exception; otherwise, any remaining successful calls are returned as normal
- **flatten** if not callable and evalues as True, updates the returned dictionary with the dictionary (instead of a nested dictionary)
- nones if not callable and evalues as True, includes entries for calls that return None (default is False)
- **traceback_delay** if *False*, immediately show traceback information on a thread exception; if *True* (the default), wait until all threads finish

Returns the values returned by each function

Return type dictionary of keyed by function

Here are some examples:

Example Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> def do_something_1 ():
>>>    time.sleep(0.5)
>>>    return 1
>>> def do_something_2 ():
>>>    time.sleep(1)
```

(continues on next page)

(continued from previous page)

```
>>> return 2
>>> rets = concurrent(myfunc1, myfunc2)
>>> rets[do_something_1]
1
```

Example To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>     time.sleep(2)
>>>     return a,b,c
>>> rets = concurrent(myfunc1, Call(myfunc3,a,b,c=c))
>>> rets[do_something_3]
a, b, c
```

Caveats

- Because the calls are in different threads, not different processes, this should be used for IO-bound functions (not CPU-intensive functions).
- Be careful about thread safety.

When the callable object is a Device method, :func concurrency: checks the Device object state.concurrency_support for compatibility before execution. If this check returns *False*, this method raises a ConcurrentException.

```
labbench.util.sequentially(*funcs, **kws)
```

Call each function or method listed in *funcs sequentially. The goal is to emulate the behavior of the *concurrently* function, with some of the same support for updating result dictionaries.

Multiple references to the same function in *funcs* only result in one call. The *catch* and *flatten* arguments may be callables, in which case they are executed (and their values are treated as defaults).

Parameters objs – each argument may be a callable (function or class that

defines a __call__ method), or context manager (such as a Device instance) :param catch: if *False* (the default), a *ConcurrentException* is raised if any of *funcs* raise an exception; otherwise, any remaining successful calls are returned as normal :param flatten: if not callable and evalues as True, updates the returned dictionary with the dictionary (instead of a nested dictionary) :param nones: if not callable and evalues as True, includes entries for calls that return None (default is False) :return: the values returned by each function :rtype: dictionary of keyed by function.

Here are some examples:

Example Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> import labbench as lb
>>> def do_something_1 ():
>>> time.sleep(0.5)
>>> return 1
>>> def do_something_2 ():
>>> time.sleep(1)
>>> return 2
>>> rets = lb.sequentially(myfunc1, myfunc2)
>>> rets[do_something_1]
1
```

Example To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>         time.sleep(2)
>>>         return a,b,c
>>>         rets = lb.sequentially(myfunc1, Call(myfunc3,a,b,c=c))
>>>         rets[do_something_3]
a, b, c
```

Because :func sequentially: does not use threading, it does not check whether a Device method supports concurrency before it runs.

```
class labbench.util.Call(func, *args, **kws)
    Bases: object
```

Wrap a function to apply arguments for threaded calls to *concurrently*. This can be passed in directly by a user in order to provide arguments; otherwise, it will automatically be wrapped inside *concurrently* to keep track of some call metadata during execution.

```
static cleanup(func_in)
```

Cleanup threading (concurrent execution only)

```
set_queue (queue)
```

```
{\tt static} {\tt setup} (func\_in)
```

Setup threading (concurrent execution only), including checks for whether a Device instance indicates it supports concurrent execution or not.

```
exception labbench.util.ConcurrentException
```

Bases: Exception

args

with_traceback()

Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.

```
class labbench.util.ConfigStore
```

Bases: object

Define dictionaries of configuration settings in subclasses of this object. Each dictionary should be an attribute of the subclass. The all() class method returns a flattened dictionary consisting of all values of these dictionary attributes, keyed according to '{attr_name}_{attr_key}', where {attr_name} is the name of the dictionary attribute and {attr_key} is the nested dictionary key.

```
classmethod all()
```

classmethod frame()

class labbench.util.ConcurrentRunner

Bases: object

Concurrently runs all staticmethods or classmethods defined in the subclass.

```
class labbench.util.FilenameDict(*args, **kws)
```

```
Bases: sortedcontainers.sorteddict.SortedDict
```

Sometimes instrument configuration file can be defined according to a combination of several test parameters.

This class provides a way of mapping these parameters to and from a filename string.

They keys are sorted alphabetically, just as in the underlying SortedDict.

clear()

Remove all items from sorted dict.

Runtime complexity: O(n)

copy()

Return a shallow copy of the sorted dict.

Runtime complexity: O(n)

Returns new sorted dict

classmethod from_filename(filename)

Convert from a FilenameDict filename string to a FilenameDict object.

classmethod from_index(df, value=None)

Make a FilenameDict where the keys are taken from df.index and the values are constant values provided.

classmethod fromkeys (iterable, value=None)

Return a new sorted dict initialized from iterable and value.

Items in the sorted dict have keys from iterable and values equal to value.

Runtime complexity: O(n*log(n))

Returns new sorted dict

get()

Return the value for key if key is in the dictionary, else default.

iloc

Cached reference of sorted keys view.

Deprecated in version 2 of Sorted Containers. Use SortedDict.keys() instead.

items()

Return new sorted items view of the sorted dict's items.

See SortedItemsView for details.

Returns new sorted items view

key

Function used to extract comparison key from keys.

Sorted dict compares keys directly when the key function is none.

keys ()

Return new sorted keys view of the sorted dict's keys.

See SortedKeysView for details.

Returns new sorted keys view

peekitem(index=-1)

Return (key, value) pair at index in sorted dict.

Optional argument *index* defaults to -1, the last item in the sorted dict. Specify index=0 for the first item in the sorted dict.

Unlike SortedDict.popitem(), the sorted dict is not modified.

If the *index* is out of range, raises IndexError.

Runtime complexity: O(log(n))

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.peekitem()
('c', 3)
>>> sd.peekitem(0)
('a', 1)
```

(continues on next page)

(continued from previous page)

```
>>> sd.peekitem(100)
Traceback (most recent call last):
...
IndexError: list index out of range
```

Parameters index (*int*) – index of item (default -1)

Returns key and value pair

Raises IndexError – if index out of range

```
pop (key, default=<not-given>)
```

Remove and return value for item identified by key.

If the key is not found then return default if given. If default is not given then raise KeyError.

Runtime complexity: O(log(n)) – approximate.

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.pop('c')
3
>>> sd.pop('z', 26)
26
>>> sd.pop('y')
Traceback (most recent call last):
...
KeyError: 'y'
```

Parameters

- **key** *key* for item
- **default** *default* value if key not found (optional)

Returns value for item

Raises KeyError – if key not found and default not given

```
popitem (index=-1)
```

Remove and return (key, value) pair at index from sorted dict.

Optional argument *index* defaults to -1, the last item in the sorted dict. Specify index=0 for the first item in the sorted dict.

If the sorted dict is empty, raises KeyError.

If the *index* is out of range, raises IndexError.

Runtime complexity: O(log(n))

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.popitem()
('c', 3)
>>> sd.popitem(0)
('a', 1)
>>> sd.popitem(100)
Traceback (most recent call last):
...
IndexError: list index out of range
```

Parameters index (int) – index of item (default -1)

Returns key and value pair

Raises

- **KeyError** if sorted dict is empty
- **IndexError** if *index* out of range

setdefault (key, default=None)

Return value for item identified by key in sorted dict.

If key is in the sorted dict then return its value. If key is not in the sorted dict then insert key with value default and return default.

Optional argument default defaults to none.

Runtime complexity: O(log(n)) – approximate.

```
>>> sd = SortedDict()
>>> sd.setdefault('a', 1)
1
>>> sd.setdefault('a', 10)
1
>>> sd
SortedDict({'a': 1})
```

Parameters

- key key for item
- **default** value for item (default None)

Returns value for item identified by key

```
update (*args, **kwargs)
```

Update sorted dict with items from args and kwargs.

Overwrites existing items.

Optional arguments *args* and *kwargs* may be a mapping, an iterable of pairs or keyword arguments. See SortedDict.__init__() for details.

Parameters

- args mapping or iterable of pairs
- **kwargs** keyword arguments mapping

values()

Return new sorted values view of the sorted dict's values.

See SortedValuesView for details.

Returns new sorted values view

```
labbench.util.hash_caller(call_depth=1)
```

Use introspection to return an SHA224 hex digest of the caller, which is almost certainly unique to the combination of the caller source code and the arguments passed it.

```
labbench.util.kill_by_name(*names)
```

Kill one or more running processes by the name(s) of matching binaries.

Parameters names (str) – list of names of processes to kill

Example

```
>>> # Kill any binaries called 'notepad.exe' or 'notepad2.exe'
>>> kill_by_name('notepad.exe', 'notepad2.exe')
```

Notes

Looks for a case-insensitive match against the Process.name() in the psutil library. Though psutil is cross-platform, the naming convention returned by name() is platform-dependent. In windows, for example, name() usually ends in '.exe'.

```
labbench.util.check master()
```

Raise ThreadEndedByMaster if the master thread as requested this thread to end.

```
labbench.util.retry(exception_or_exceptions, tries=4, delay=0, backoff=0, excep-
tion_func=<function <lambda>>)
```

This decorator causes the function call to repeat, suppressing specified exception(s), until a maximum number of retries has been attempted. - If the function raises the exception the specified number of times, the underlying exception is raised. - Otherwise, return the result of the function call.

Example

The following retries the telnet connection 5 times on ConnectionRefusedError:

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Parameters

- exception_or_exceptions Exception (sub)class (or tuple of exception classes) to watch for
- **tries** (*int*) number of times to try before giving up
- **delay** (float) initial delay between retries in seconds
- backoff (float) backoff to multiply to the delay for each retry
- exception_func (callable) function to call on exception before the next retry

```
labbench.util.show_messages(minimum_level)
```

Configure screen debug message output for any messages as least as important as indicated by level.

Parameters minimum_level – One of 'debug', 'warning', 'error', or None. If None, there will be no output.

Returns None

```
labbench.util.sleep (seconds, tick=1.0)
```

Drop-in replacement for time.sleep that raises ConcurrentException if another thread requests that all threads stop.

```
labbench.util.stopwatch(desc=")
```

Time a block of code using a with statement like this:

```
>>> with stopwatch('sleep statement'):
>>>    time.sleep(2)
sleep statement time elapsed 1.999s.
```

Parameters desc(str) – text for display that describes the event being timed

Returns context manager

```
class labbench.util.Testbed(config=None, concurrent=True)
    Bases: object
```

Base class for testbeds that contain multiple Device instances or other objects like database managers that implement context management.

Use a with block with the testbed instance to connect everything at once like so:

```
with Testbed() as testbed:
    # use the testbed here
    pass
```

or optionally connect only a subset of devices like this:

```
testbed = Testbed()
with testbed.dev1, testbed.dev2:
    # use the testbed.dev1 and testbed.dev2 here
pass
```

Make your own subclass of Testbed with a custom *make* method to define the Device or database manager instances, and a custom *startup* method to implement custom code to set up the testbed after each Device is connected.

cleanup()

This is called automatically immediately before disconnect if the testbed is connected using the *with* statement block.

Implement any custom code here in Testbed subclasses to implement startup of the testbed given connected Device instances.

make()

Implement this method in a subclass of Testbed. It should the drivers as attributes of the Testbed instance, for example:

```
self.dev1 = MyDevice()
```

This is called automatically when when the testbed class is instantiated.

startup()

This is called automatically after connect if the testbed is connected using the with statement block.

Implement any custom code here in Testbed subclasses to implement startup of the testbed given connected Device instances.

```
class labbench.util.ThreadSandbox(factory, should_sandbox_func=None)
    Bases: object
exception labbench.util.ThreadEndedByMaster
```

Bases: RuntimeError

args

```
with_traceback()
```

Exception.with_traceback(tb) - set self.__traceback__ to tb and return self.

```
labbench.util.until_timeout (exception_or_exceptions, timeout, delay=0, backoff=0, exception func=<function <lambda>>)
```

This decorator causes the function call to repeat, suppressing specified exception(s), until the specified timeout period has expired. - If the timeout expires, the underlying exception is raised. - Otherwise, return the result of the function call.

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Example

The following retries the telnet connection for 5 seconds on ConnectionRefusedError:

```
import telnetlib

@until_timeout(ConnectionRefusedError, 5)
def connect(host, port):
    t = telnetlib.Telnet()
    t.open(host,port,5)
    return t
```

Parameters

- exception_or_exceptions Exception (sub)class (or tuple of exception classes) to watch for
- **timeout** (float) time in seconds to continue calling the decorated function while suppressing exception_or_exceptions
- **delay** (float) initial delay between retries in seconds
- backoff (float) backoff to multiply to the delay for each retry
- exception_func (callable) function to call on exception before the next retry

CHAPTER 8

labbench.version module

CHAF	ТΕ	D	Q
CHAL	ĺ□	Ħ	J

Module contents

Python Module Index

labbench, 69 labbench.backends, 3 labbench.core, 27 labbench.data, 35 labbench.host, 47 labbench.notebooks, 55 labbench.util, 57 labbench.version, 67

all()labbench.util.ConfigStore class method, 59 append()labbench.data.StatesToCSV method, 39	baud_ratelabbench.backends.SerialDevice.settings at- tribute, 13
append()labbench.data.StatesToRelationalTable method,	baud_ratelabbench.backends.SerialLoggingDevice.settings
36	attribute, 16
append()labbench.data.StatesToSQLite method, 41	binary_pathlabbench.backends.CommandLineWrapper.settings
argslabbench.core.CommandNotImplementedError	attribute, 5
attribute, 33	Boolclass in labbench.core, 29
argslabbench.core.ConnectionError attribute, 27	Bytesclass in labbench.core, 28
argslabbench.core.DeviceConnectionLost attribute, 27	
argslabbench.core.DeviceException attribute, 27	Callclass in labbench.util, 59
argslabbench.core.DeviceFatalError attribute, 27	CaselessBytesEnumclass in labbench.core, 28
argslabbench.core.DeviceNotReady attribute, 27	CaselessStrEnumclass in labbench.core, 30
argslabbench.core.DeviceStateError attribute, 28	check_master()in module labbench.util, 63
argslabbench.util.ConcurrentException attribute, 59	childrenlabbench.notebooks.panel attribute, 55
argslabbench.util.ThreadEndedByMaster attribute, 64	class_init()labbench.core.Dict method, 30
arguments labbench. backends. Command Line Wrapper. setting the setting of the	nglass_init()labbench.core.List method, 29
attribute, 5	cleanup()labbench.util.Call static method, 59
$arguments_minlabbench.backends.CommandLineWrapper$	selennup()labbench.util.Testbed method, 64
attribute, 5	clear()labbench.backends.CommandLineWrapper
	method, 4
backendlabbench.backends.CommandLineWrapper	clear() labbench. backends. Labview Socket Interface
attribute, 3	method, 10
backendlabbench.backends.DotNetDevice attribute, 6	clear()labbench.backends.SerialLoggingDevice method,
backendlabbench.backends.EmulatedVISADevice at-	15
tribute, 8	clear()labbench.data.StatesToCSV method, 39
backendlabbench.backends.LabviewSocketInterface	clear()labbench.data.StatesToRelationalTable method, 37
attribute, 10	clear()labbench.data.StatesToSQLite method, 42
backendlabbench.backends.SerialDevice attribute, 12	clear()labbench.util.FilenameDict method, 59
backendlabbench.backends.SerialLoggingDevice at-	close()labbench.data.StatesToCSV method, 39
tribute, 15	close()labbench.data.StatesToRelationalTable method, 37
backendlabbench.backends.TelnetDevice attribute, 18	close()labbench.data.StatesToSQLite method, 42
backendlabbench.backends.VISADevice attribute, 20	com_objectlabbench.backends.Win32ComDevice.settings
backendlabbench.backends.Win32ComDevice attribute,	attribute, 25
24	CommandLineWrapperclass in labbench.backends, 3
backendlabbench.core.Device attribute, 31	CommandLineWrapper.settingsclass in
backendlabbench.host.Email attribute, 50	labbench.backends, 4
backendlabbench.host.Host attribute, 47	CommandLineWrapper.stateclass in labbench.backends,
backendlabbench.host.LogStderr attribute, 51	5
background()labbench.backends.CommandLineWrapper	CommandNotImplementedError, 33
method, 3	Complexclass in labbench.core, 28

concurrency supportlabbench.backends.CommandLineWrappansettidlabbench.backends.EmulatedVISADevice.state attribute, 5 attribute. 9 concurrency supportlabbench.backends.DotNetDevice.settingnnectedlabbench.backends.LabviewSocketInterface.state attribute, 7 attribute, 11 concurrency supportlabbench.backends.EmulatedVISADevicennettingkabbench.backends.SerialDevice.state attribute, attribute, 9 concurrency supportlabbench.backends.LabviewSocketIntecfarmexettidilgbbench.backends.SerialLoggingDevice.state attribute, 11 attribute, 18 concurrency supportlabbench.backends.SerialDevice.settingconnectedlabbench.backends.TelnetDevice.state atattribute, 13 tribute, 20 concurrency_supportlabbench.backends.SerialLoggingDevicennttintedlabbench.backends.VISADevice.state attribute, attribute, 16 $concurrency_support labbench. backends. Telnet Device. setting \textbf{\o} nnected labbench. backends. Win 32 Com Device. state - at-least one of the concurrency of the c$ attribute, 19 tribute, 25 concurrency_supportlabbench.backends.VISADevice.settingconnectedlabbench.core.Device.state attribute, 32 attribute, 22 connectedlabbench.host.Email.state attribute, 51 concurrency_supportlabbench.backends.Win32ComDevice.settingstedlabbench.host.Host.state attribute, 49 connectedlabbench.host.LogStderr.state attribute, 53 attribute, 25 concurrency supportlabbench.core.Device.settings ConnectionError, 27 attribute, 32 copy()labbench.util.FilenameDict method, 59 concurrency_supportlabbench.host.Email.settings atdata_formatlabbench.backends.SerialLoggingDevice.settings tribute, 50 attribute, 16 concurrency_supportlabbench.host.Host.settings atdefault valuelabbench.core.Bool attribute, 29 tribute, 48 default_valuelabbench.core.List attribute, 29 concurrency supportlabbench.host.LogStderr.settings atdefault valuelabbench.core.Unicode attribute, 28 tribute, 52 default value repr()labbench.core.Dict method, 30 ConcurrentException, 59 default_value_repr()labbench.core.List method, 29 concurrently()in module labbench.util, 57 define()labbench.backends.CommandLineWrapper.settings ConcurrentRunnerclass in labbench.util, 59 class method, 5 ConfigStoreclass in labbench.util, 59 define()labbench.backends.DotNetDevice.settings class configure()labbench.backends.SerialLoggingDevice method. 7 method, 15 define()labbench.backends.EmulatedVISADevice.settings connect()labbench.backends.CommandLineWrapper class method, 9 method, 4 define()labbench.backends.LabviewSocketInterface.settings connect()labbench.backends.DotNetDevice method, 6 class method, 11 connect()labbench.backends.EmulatedVISADevice define()labbench.backends.SerialDevice.settings class method, 8 method, 13 connect()labbench.backends.LabviewSocketInterface define()labbench.backends.SerialLoggingDevice.settings method, 10 class method, 16 connect()labbench.backends.SerialDevice method, 12 define()labbench.backends.TelnetDevice.settings class connect()labbench.backends.SerialLoggingDevice method, 19 method, 15 define()labbench.backends.VISADevice.settings class connect()labbench.backends.TelnetDevice method, 18 method, 22 connect()labbench.backends.VISADevice method, 20 define()labbench.backends.Win32ComDevice.settings connect()labbench.backends.Win32ComDevice method, class method, 25 24 define()labbench.core.Device.settings class method, 32 connect()labbench.core.Device method, 31 define()labbench.host.Email.settings class method, 50 connect()labbench.host.Email method, 50 define()labbench.host.Host.settings class method, 48 connect()labbench.host.Host method, 47 define()labbench.host.LogStderr.settings class method, 52 connect()labbench.host.LogStderr method, 52 delaylabbench.backends.LabviewSocketInterface atconnectedlabbench.backends.CommandLineWrapper.state tribute, 10 attribute, 6 devalidate()labbench.core.CaselessBytesEnum method, connectedlabbench.backends.DotNetDevice.state

Index 73

tribute, 7

Device class in labbench.core, 30	failure_messagelabbench.host.Email.settings attribute	
Device.settingsclass in labbench.core, 31	fetch()labbench.backends.SerialLoggingDevice meth	10u,
Device.stateclass in labbench.core, 32	15	
DeviceConnectionLost, 27	FilenameDictclass in labbench.util, 59	
DeviceException, 27	Floatclass in labbench.core, 28	
DeviceFatalError, 27	flush()labbench.host.LogStderr method, 52	
DeviceNotReady, 27	foreground()labbench.backends.CommandLineWrapp	er
deviceslabbench.notebooks.panel attribute, 55	method, 4	
DeviceStateError, 28	frame()labbench.util.ConfigStore class method, 59	
Dictclass in labbench.core, 29		lass
disconnect()labbench.backends.CommandLineWrapper	method, 60	
method, 4	_ "	lass
disconnect()labbench.backends.DotNetDevice method, 6	method, 13	
disconnect()labbench.backends.EmulatedVISADevice	from_hwid()labbench.backends.SerialLoggingDevice	
method, 8	class method, 15	
disconnect()labbench.backends.LabviewSocketInterface	from_index()labbench.util.FilenameDict class method	
method, 10	fromkeys()labbench.util.FilenameDict class method, 6	50
disconnect()labbench.backends.SerialDevice method, 12		
disconnect()labbench.backends.SerialLoggingDevice method, 15	generatorslabbench.backends.EmulatedVISADevice attribute, 8	
disconnect()labbench.backends.TelnetDevice method, 18	get()labbench.data.StateAggregator method, 35	
disconnect()labbench.backends.VISADevice method, 21	get()labbench.data.StatesToCSV method, 39	
disconnect()labbench.backends.Win32ComDevice	get()labbench.data.StatesToRelationalTable method, 3	37
method, 24	get()labbench.data.StatesToSQLite method, 42	
disconnect()labbench.core.Device method, 31	get()labbench.util.FilenameDict method, 60	
disconnect()labbench.host.Email method, 50	get_git_browse_urllabbench.host.Host attribute, 47	
disconnect()labbench.host.Host method, 47	get_git_commitlabbench.host.Host attribute, 47	
disconnect()labbench.host.LogStderr method, 52	get_git_remote_urllabbench.host.Host attribute, 47	
dll_namelabbench.backends.DotNetDevice attribute, 6	get_hostnamelabbench.host.Host attribute, 47	
doc_attrslabbench.core.CaselessBytesEnum attribute, 29	get_loglabbench.host.Host attribute, 47	
doc_attrslabbench.core.CaselessStrEnum attribute, 30	get_timelabbench.host.Host attribute, 47	
doc_attrslabbench.core.Float attribute, 28	getter()labbench.backends.CommandLineWrapper.sta	ite
doc_attrslabbench.core.Int attribute, 28	class method, 6	
DotNetDeviceclass in labbench.backends, 6	getter()labbench.backends.DotNetDevice.state c	lass
DotNetDevice.settingsclass in labbench.backends, 7	method, 8	
DotNetDevice.stateclass in labbench.backends, 7 dsrdtrlabbench.backends.SerialDevice.settings attribute,	getter()labbench.backends.EmulatedVISADevice.state class method, 9	e
1/	getter()labbench.backends.LabviewSocketInterface.sta	ate
dsrdtrlabbench.backends.SerialLoggingDevice.settings	class method, 11	
		lass
attribute, 17	method, 14	1400
element_error()labbench.core.Dict method, 30	getter()labbench.backends.SerialLoggingDevice.state	
element_error()labbench.core.List method, 29	class method, 18	
Emailclass in labbench.host, 49		lass
Email.settingsclass in labbench.host, 50	method, 20	1433
Email.stateclass in labbench.host, 51		lass
EmulatedVISADeviceclass in labbench.backends, 8	method, 23	1433
Emulated VISA Device setting sclass in		lass
labbench.backends, 8	method, 25	1433
Emulated VISA Device. state class in labbench. backends, 9	getter()labbench.core.Device.state class method, 32	
error()labbench.core.Dict method, 30	getter()labbench.host.Email.state class method, 52	
error()labbench.core.List method, 29	getter()labbench.host.Host.state class method, 49	
exception_on_stderrlabbench.backends.CommandLineWra		3
attribute, 4	git_browse_urllabbench.host.Host.state attribute, 49	,
autionic, i	mil did it de di liudo de li di li dobi, li dobi, di li di l	

git_commit_idlabbench.host.Host.state attribute, 49 git_remote_urllabbench.host.Host.state attribute, 49	LabviewSocketInterface.settingsclass labbench.backends, 10	in
	LabviewSocketInterface.stateclass in labbench.bad	ckends,
hash_caller()in module labbench.util, 62	11	
Hostclass in labbench.host, 47	length_error()labbench.core.List method, 29	
Host.settingsclass in labbench.host, 47	librarylabbench.backends.DotNetDevice attribute,	6
Host.stateclass in labbench.host, 48	linspace()in module labbench.notebooks, 56	
hostnamelabbench.host.Host.state attribute, 49	Listclass in labbench.core, 29	
	list_devices()in module labbench.core, 33	
identitylabbench.backends.EmulatedVISADevice.state attribute, 9	list_ports()labbench.backends.SerialDevice method, 13	static
identitylabbench.backends.VISADevice.state attribute, 23	list_ports()labbench.backends.SerialLoggingDevic static method, 16	ee
iloclabbench.util.FilenameDict attribute, 60	list_resources()labbench.backends.VISADevice	class
index_labellabbench.data.StatesToCSV attribute, 39	method, 21	Class
index_labellabbench.data.StatesToRelationalTable at-	loglabbench.host.Host.state attribute, 49	
tribute, 37		
index_labellabbench.data.StatesToSQLite attribute, 42	loglabbench.host.LogStderr attribute, 52	
info()labbench.core.CaselessStrEnum method, 30	log_formatlabbench.host.Host attribute, 47	
info()labbench.core.Dict method, 30	log_progress()in module labbench.notebooks, 55	
info()labbench.core.List method, 29	LogStderrclass in labbench.host, 51	
info_textlabbench.core.Bool attribute, 29	LogStderr.settingsclass in labbench.host, 52	
info_textlabbench.core.Bytes attribute, 28	LogStderr.stateclass in labbench.host, 52	
info_textlabbench.core.CaselessBytesEnum attribute, 29		
info_textlabbench.core.Complex attribute, 28	make()labbench.util.Testbed method, 64	1 20
	make_dynamic_default()labbench.core.Dict metho	
info_textlabbench.core.Float attribute, 28	make_dynamic_default()labbench.core.List metho	
info_textlabbench.core.Int attribute, 28	master_filenamelabbench.data.StatesToSQLite at	tribute,
info_textlabbench.core.TCPAddress attribute, 30	42	
info_textlabbench.core.Unicode attribute, 28	max_queue_sizelabbench.backends.SerialLogging	Device.setting
instance_init()labbench.core.Dict method, 30	attribute, 17	
instance_init()labbench.core.List method, 29	metadata()labbench.host.Host method, 47	
Intclass in labbench.core, 28		
items()labbench.util.FilenameDict method, 60	ncolslabbench.notebooks.panel attribute, 55	
keylabbench.util.FilenameDict attribute, 60	no_state_argumentslabbench.backends.Commandlattribute, 4	LineWrapper
key()labbench.data.StateAggregator method, 35	nonscalar_file_typelabbench.data.StatesToCSV at	tribute,
key()labbench.data.StatesToCSV method, 39	40	
key()labbench.data.StatesToRelationalTable method, 37		
key()labbench.data.StatesToSQLite method, 42	observe()labbench.data.StateAggregator method, 3	35
keys()labbench.util.FilenameDict method, 60	observe()labbench.data.StatesToCSV method, 40	
$\label{linewrapper} kill () labbench. backends. Command Line Wrapper\ method, \\ 4$	observe()labbench.data.StatesToRelationalTable n	nethod,
kill_by_name()in module labbench.util, 62	observe()labbench.data.StatesToSQLite method, 4	2
klasslabbench.core.Dict attribute, 30	observe_settings()labbench.data.StateAggregator	
klasslabbench.core.List attribute, 29	method, 35	
	observe_settings()labbench.data.StatesToCSV n	nethod,
labbenchmodule, 69	40	
labbench.backendsmodule, 3	observe_settings()labbench.data.StatesToRelationa	alTable
labbench.coremodule, 27	method, 37	
labbench.datamodule, 35	observe_settings()labbench.data.StatesToSQLite n	nethod,
labbench.hostmodule, 47	42	*
labbench.notebooksmodule, 55	observe_states()labbench.data.StateAggregator n	nethod.
labbench.utilmodule, 57	35	,
labbench.versionmodule, 67	observe_states()labbench.data.StatesToCSV metho	od. 40
LabviewSocketInterfaceclass in labbench.backends, 10		,

observe states()labbench.data.StatesToRelationalTable resourcelabbench.backends.VISADevice.settings method, 37 tribute, 22 observe states()labbench.data.StatesToSQLite resourcelabbench.backends.Win32ComDevice.settings method. attribute, 25 open()labbench.data.StatesToCSV method, 40 resourcelabbench.core.Device.settings attribute, 32 open()labbench.data.StatesToRelationalTable method, 38 resourcelabbench.host.Email.settings attribute, 51 open()labbench.data.StatesToSOLite method, 43 resourcelabbench.host.Host.settings attribute, 48 optionslabbench.backends.EmulatedVISADevice.state resourcelabbench.host.LogStderr.settings attribute, 52 respawnlabbench.backends.CommandLineWrapper attribute, 10 optionslabbench.backends.VISADevice.state attribute, 23 attribute, 4 overlap_and_block()labbench.backends.VISADevice retry()in module labbench.util, 63 method, 21 rtsctslabbench.backends.SerialDevice.settings panelclass in labbench.notebooks, 55 rtsctslabbench.backends.SerialLoggingDevice.settings paritylabbench.backends.SerialDevice.settings attribute, attribute, 17 running()labbench.backends.CommandLineWrapper paritylabbench.backends.SerialLoggingDevice.settings method, 4 attribute, 17 running()labbench.backends.SerialLoggingDevice peekitem()labbench.util.FilenameDict method, 60 method, 16 poll ratelabbench.backends.SerialLoggingDevice.settings rx buffer sizelabbench.backends.LabviewSocketInterface attribute, 17 attribute, 10 pop()labbench.util.FilenameDict method, 61 rx portlabbench.backends.LabviewSocketInterface popitem()labbench.util.FilenameDict method, 61 attribute, 10 portlabbench.backends.TelnetDevice.settings attribute, 19 preset()labbench.backends.VISADevice method, 21 send_summary()labbench.host.Email method, 50 senderlabbench.host.Email.settings attribute, 51 query()labbench.backends.VISADevice method, 21 sequentially()in module labbench.util, 58 SerialDeviceclass in labbench.backends, 12 range()in module labbench.notebooks, 55 SerialDevice.settingsclass in labbench.backends, 13 read()in module labbench.data, 44 SerialDevice.stateclass in labbench.backends, 14 read()labbench.backends.LabviewSocketInterface SerialLoggingDeviceclass in labbench.backends, 15 method, 10 SerialLoggingDevice.settingsclass in labbench.backends, read_relational()in module labbench.data, 44 read_stdout()labbench.backends.CommandLineWrapper SerialLoggingDevice.stateclass in labbench.backends, 17 method, 4 set backend()labbench.backends.EmulatedVISADevice read terminationlabbench.backends.VISADevice.settings class method, 8 attribute, 22 set_backend()labbench.backends.VISADevice recipientslabbench.host.Email.settings attribute, 50 method, 21 $resource labbench. backends. Command Line Wrapper. settings\\set_device_labels\\() labbench. data. State Aggregator$ attribute, 5 method, 36 resourcelabbench.backends.DotNetDevice.settings set device labels()labbench.data.StatesToCSV method, attribute, 7 resourcelabbench.backends.EmulatedVISADevice.settings set device labels()labbench.data.StatesToRelationalTable attribute. 9 method, 38 resource labbench. backends. Labview Socket Interfaceset device labels()labbench.data.StatesToSQLite attribute, 10 method, 43 $resource labbench. backends. Labview Socket Interface. setting \\ \underbrace{set_path_format() labbench. data. States ToCSV\ method, 41}_{note that the control of t$ attribute, 11 set path format()labbench.data.StatesToRelationalTable resourcelabbench.backends.SerialDevice.settings method, 38 tribute, 14 set path format()labbench.data.StatesToSOLite method, resourcelabbench.backends.SerialLoggingDevice.settings attribute, 17 set queue()labbench.util.Call method, 59 resourcelabbench.backends.TelnetDevice.settings set_relational_file_format()labbench.data.StatesToCSV tribute, 19 method, 41

at-

class

set_relational_file_format()labbench.data.StatesToRelational			
method, 38	tribute, 14		
$set_relational_file_format() labbench. data. States To SQLite$	stopbitslabbench.backends.SerialLoggingDevice.settings		
method, 43	attribute, 17		
$set_row_preprocessor() labbench. data. States To CSV$	stopwatch()in module labbench.util, 63		
method, 41	success_messagelabbench.host.Email.settings attribute,		
set_row_preprocessor()labbench.data.StatesToRelationalTa method, 38	ble 51		
set_row_preprocessor()labbench.data.StatesToSQLite	table_namelabbench.data.StatesToSQLite attribute, 44		
method, 43	TCPAddressclass in labbench.core, 30		
setdefault()labbench.util.FilenameDict method, 62	TelnetDeviceclass in labbench.backends, 18		
setter()labbench.backends.CommandLineWrapper.state	TelnetDevice.settingsclass in labbench.backends, 19		
class method, 6	TelnetDevice.stateclass in labbench.backends, 19		
setter()labbench.backends.DotNetDevice.state class	Testbedclass in labbench.util, 64		
method, 8	ThreadEndedByMaster, 64		
setter()labbench.backends.EmulatedVISADevice.state	ThreadSandboxclass in labbench.util, 64		
class method, 10	timelabbench.host.Host.state attribute, 49		
setter()labbench.backends.LabviewSocketInterface.state	time_formatlabbench.host.Host attribute, 49		
class method, 12	time out labbench. backends. Command Line Wrapper. settings		
setter()labbench.backends.SerialDevice.state class	attribute, 5		
method, 14	timeoutlabbench.backends.LabviewSocketInterface		
setter()labbench.backends.SerialLoggingDevice.state	attribute, 12		
class method, 18	timeoutlabbench.backends.SerialDevice.settings at-		
setter()labbench.backends.TelnetDevice.state class	tribute, 14		
method, 20	timeoutlabbench.backends.SerialLoggingDevice.settings		
setter()labbench.backends.VISADevice.state class	attribute, 17		
method, 23	timeoutlabbench.backends.TelnetDevice.settings at-		
setter()labbench.backends.Win32ComDevice.state class	tribute, 19		
method, 26	to_feather()in module labbench.data, 45		
setter()labbench.core.Device.state class method, 32	tx_portlabbench.backends.LabviewSocketInterface		
setter()labbench.host.Email.state class method, 51	attribute, 12		
setter()labbench.host.Host.state class method, 49			
setter()labbench.host.LogStderr.state class method, 53	Unicodeclass in labbench.core, 28		
setup()labbench.data.StatesToCSV method, 41	until_timeout()in module labbench.util, 65		
setup()labbench.data.StatesToRelationalTable method, 38	update()labbench.util.FilenameDict method, 62		
setup()labbench.data.StatesToSQLite method, 44	validate()labbench.core.Bool method, 29		
setup()labbench.util.Call static method, 59	validate()labbench.core.Bytes method, 28		
show_messages()in module labbench.util, 63	validate()labbench.core.CaselessBytesEnum method, 29		
sleep()in module labbench.util, 63	validate()labbench.core.CaselessBytesEntin method, 29		
start()labbench.backends.SerialLoggingDevice method,	validate()labbench.core.Complex method, 28		
17	validate()labbench.core.Dict method, 30		
startup()labbench.util.Testbed method, 64	validate()labbench.core.Float method, 28		
StateAggregatorclass in labbench.data, 35	validate()labbench.core.Int method, 28		
StatesToCSVclass in labbench.data, 39	validate()labbench.core.List method, 29		
StatesToRelationalTableclass in labbench.data, 36	validate()labbench.core.TCPAddress method, 30		
StatesToSQLiteclass in labbench.data, 41	validate()labbench.core.Unicode method, 28		
$status_bytel abbench. backends. Emulated VISA Device. state$	validate_elements()labbench.core.Dict method, 30		
attribute, 10	validate_elements()labbench.core.List method, 29		
status_bytelabbench.backends.VISADevice.state at-	values()labbench.util.FilenameDict method, 62		
tribute, 23	VISADeviceclass in labbench.backends, 20		
stop()labbench.backends.SerialLoggingDevice method,	VISADevice.settingsclass in labbench.backends, 22		
18	VICAD. See Astrological Islands Islands and Co.		
stop_timeoutlabbench.backends.SerialLoggingDevice.settingSSADevice.suppress_timeoutclassin			
attribute, 17	labbench.backends, 23		

```
VISADevice.VISADeviceclass in labbench.backends, 20
wait()labbench.backends.VISADevice method, 24
widgetlabbench.notebooks.panel attribute, 55
Win32ComDeviceclass in labbench.backends, 24
Win32ComDevice.settingsclass in labbench.backends, 24
Win32ComDevice.stateclass in labbench.backends, 25
with\_traceback() labbench.core. Command Not Implemented Error
         method, 33
with traceback()labbench.core.ConnectionError method,
with traceback()labbench.core.DeviceConnectionLost
         method, 28
with traceback()labbench.core.DeviceException method,
with_traceback()labbench.core.DeviceFatalError method,
with_traceback()labbench.core.DeviceNotReady method,
with traceback()labbench.core.DeviceStateError method,
with_traceback()labbench.util.ConcurrentException
         method, 59
with\_traceback() labbench.util. ThreadEndedBy Master
         method, 65
write()labbench.backends.LabviewSocketInterface
         method, 12
write()labbench.backends.VISADevice method, 24
write()labbench.data.StatesToCSV method, 41
write()labbench.data.StatesToRelationalTable method, 38
write()labbench.data.StatesToSQLite method, 44
write()labbench.host.LogStderr method, 53
write_stdin()labbench.backends.CommandLineWrapper
         method, 6
write_terminationlabbench.backends.SerialDevice.settings
         attribute, 14
write_terminationlabbench.backends.SerialLoggingDevice.settings
         attribute, 17
write_terminationlabbench.backends.VISADevice.settings
         attribute, 22
xonxofflabbench.backends.SerialDevice.settings
         tribute, 14
xonxofflabbench.backends.SerialLoggingDevice.settings
         attribute, 17
```