# aat Documentation

***Release 0.1.0***

**Tim Paine**

**Oct 19, 2020**

# Contents

Build Status Coverage License PyPI Docs

`aat` is an asynchronous, event-driven framework for writing algorithmic trading strategies in python with optional acceleration in C++. It is designed to be modular and extensible, with support for a wide variety of instruments and strategies, live trading across (and between) multiple exchanges, fully integrated backtesting support, slippage and transaction cost modeling, and robust reporting and risk mitigation through manual and programatic algorithm controls.

Like Zipline and Lean, `aat` exposes a single strategy class which is utilized for both live trading and backtesting. The strategy class is simple enough to write and test algorithms quickly, but extensible enough to allow for complex slippage and transaction cost modeling, as well as mid- and post- trade analysis.

`aat` is in active use for live algorithmic trading on equities, commodity futures contracts, and commodity futures spreads by undisclosed funds.

CHAPTER 1

# Internals

`aat`'s engine is composed of 4 major parts.

- trading engine
- risk management engine
- execution engine
- backtest engine

## 1.1 Trading Engine

The trading engine initializes all exchanges and strategies, then martials data, trade requests, and trade responses between the strategy, risk, execution, and exchange objects, while keeping track of high-level statistics on the system

## 1.2 Risk Management Engine

The risk management engine enforces trading limits, making sure that stategies are limited to certain risk profiles. It can modify or remove trade requests prior to execution depending on user preferences and outstanding positions and orders.

## 1.3 Execution engine

The execution engine is a simple passthrough to the underlying exchanges. It provides a unified interface for creating various types of orders.

## 1.4 Backtest engine

The backtest engine provides the ability to run the same stragegy offline against historical data.

# CHAPTER 2

# Core Components

`aat` has a variety of core classes and data structures, the most important of which are the `Strategy` and `Exchange` classes.

## 2.1 Trading Strategy

The core element of `aat` is the trading strategy interface. It includes both data processing and order management functionality. Users subclass this class in order to implement their strategies. Methods of the form `onNoun` are used to handle market data events, while methods of the form `onVerb` are used to handle order entry events. There are also a variety of order management and data subscription methods available.

The only method that is required to be implemented is the `onTrade` method. The full specification of a strategy is given here (we will look at an example below).

```
class Strategy(metaclass=ABCMeta):
    ########################
    # Event Handler Methods #
    ########################
    @abstractmethod
    async def onTrade(self, event: Event):
        '''Called whenever a `Trade` event is received'''

    async def onOrder(self, event: Event) -> None:
        '''Called whenever an Order `Open`, `Cancel`, `Change`, or `Fill` event is␣
↪received'''
        pass

    async def onOpen(self, event: Event):
        '''Called whenever an Order `Open` event is received'''

    async def onFill(self, event: Event):
        '''Called whenever an Order `Fill` event is received'''
```

(continues on next page)

(continued from previous page)

```
    async def onCancel(self, event: Event):
        '''Called whenever an Order `Cancel` event is received'''

    async def onChange(self, event: Event):
        '''Called whenever an Order `Change` event is received'''

    async def onError(self, event: Event):
        '''Called whenever an internal error occurs'''

    async def onStart(self):
        '''Called once at engine initialization time'''

    async def onExit(self):
        '''Called once at engine exit time'''

    async def onHalt(self, data):
        '''Called whenever an exchange `Halt` event is received, i.e. an event to␣
↪stop trading'''

    async def onContinue(self, data):
        '''Called whenever an exchange `Continue` event is received, i.e. an event to␣
↪continue trading'''

    ##########################
    # Order Entry Callbacks #
    ##########################
    async def onBought(self, event: Event):
        '''Called on my order bought'''
        pass

    async def onSold(self, event: Event):
        '''Called on my order sold'''
        pass

    async def onTraded(self, event: Event):
        '''Called on my order bought or sold'''
        pass

    async def onRejected(self, event: Event):
        '''Called on my order rejected'''
        pass

    async def onCanceled(self, event: Event):
        '''Called on my order canceled'''
        pass

    ########################
    # Order Entry Methods #
    ########################
    async def newOrder(self, order: Order):
        '''helper method, defers to buy/sell'''

    async def cancelOrder(self, order: Order):
        '''cancel an open order'''

    async def buy(self, order: Order):
        '''submit a buy order. Note that this is merely a request for an order, it␣
↪provides no guarantees that the order will
```

(continues on next page)

(continued from previous page)

```
        execute. At a later point, if your order executes, you will receive an alert␣
↪via the `bought` method'''

    async def sell(self, order: Order):
        '''submit a sell order. Note that this is merely a request for an order, it␣
↪provides no guarantees that the order will
        execute. At a later point, if your order executes, you will receive an alert␣
↪via the `sold` method'''

    async def cancelAll(self, instrument: Instrument = None):
        '''cancel all open orders'''

    async def closeAll(self, instrument: Instrument = None):
        '''close all open positions'''

    def orders(self, instrument: Instrument = None, exchange: ExchangeType = None,␣
↪side: Side = None):
        '''select all open orders'''

    def pastOrders(self, instrument: Instrument = None, exchange: ExchangeType = None,
↪ side: Side = None):
        '''select all past orders'''

    def trades(self, instrument: Instrument = None, exchange: ExchangeType = None,␣
↪side: Side = None):
        '''select all past trades'''

    def accounts(self) -> List:
        '''get accounts from source'''

    ################
    # Risk Methods #
    ################
    def positions(self, instrument: Instrument = None, exchange: ExchangeType = None,␣
↪side: Side = None):
        '''select all positions'''

    def risk(self, position=None):
        '''Get risk metrics'''

    def priceHistory(self, instrument: Instrument):
        '''Get price history for an asset'''

    #################
    # Other Methods #
    #################
    def now(self):
        '''Return the current datetime. Useful to avoid code changes between
        live trading and backtesting. Defaults to `datetime.now`'''

    def instruments(self, type=None, exchange=None):
        '''Return list of all available instruments'''

    def exchanges(self, instrument_type=None):
        '''Return list of all available exchanges'''

    def subscribe(self, instrument=None):
```

(continues on next page)

(continued from previous page)

```
        '''Subscribe to market data for the given instrument'''

    def lookup(self, instrument):
        '''lookup an instrument on the exchange'''
```

## 2.2 Example Strategy

Here is a simple trading strategy that buys once and holds.

```
from aat import Strategy, Event, Order, Trade, Side

class BuyAndHoldStrategy(Strategy):
    def __init__(self, *args, **kwargs) -> None:
        super(BuyAndHoldStrategy, self).__init__(*args, **kwargs)

    async def onTrade(self, event: Event) -> None:
        '''Called whenever a `Trade` event is received'''
        trade: Trade = event.target

        # no past trades, no current orders
        if not self.orders(trade.instrument) and not self.trades(trade.instrument):
            req = Order(side=Side.BUY,
                        price=trade.price,
                        volume=1,
                        instrument=trade.instrument,
                        order_type=Order.Types.MARKET,
                        exchange=trade.exchange)

            print("requesting buy : {}".format(req))
            await self.newOrder(req)

    async def onBought(self, event: Event) -> None:
        trade: Trade = event.target
        print('bought {:.2f} @ {:.2f}'.format(trade.volume, trade.price))

    async def onRejected(self, event: Event) -> None:
        print('order rejected')
        import sys
        sys.exit(0)

    async def onExit(self, event: Event) -> None:
        print('Finishing...')
```

Trading strategies have only one required method handling messages:

- onTrade: Called when a trade occurs

There are other optional callbacks for more granular processing:

- onOrder: Called whenever a new order occurs, an order is filled, an order is cancelled, or an order is modified (includes the behavior of onOpen, onFill, onCancel, and onChange)
- onOpen: Called when a new order occurs
- onFill: Called when an order is filled
- onCancel: Called when an order is cancelled

- onChange: Called when an order is modified
- onError: Called when a system error occurs
- onHalt: Called when trading is halted
- onContinue: Called when trading continues
- onStart: Called when the program starts
- onExit: Called when the program shuts down

There are several callbacks for order entry:

- onTraded: called when a strategy's order is bought or sold
- onBought: called when a strategy's order is bought
- onSold: called when a strategy's order is sold
- onRejected: called when a strategy's order is rejected
- onCanceled: called when a strategy's order is canceled

There are several methods for order entry and data subscriptions...

- subscribe: subscribe to an instrument/exchange data
- instruments: get available instruments
- exchanges: get available exchanges
- lookup: lookup an instrument on the exchange
- newOrder: submit a new order
- buy (alias of newOrder): submit a new order
- sell (alias of newOrder): submit a new order
- orders: get open orders
- pastOrders: get past orders
- trades: get past trades

... several helpers for analyzing positions and risk ...

- accounts: get account information
- positions: get position information
- risk: get risk information

... and some general utility methods ...

- tradingType: get the trading type of the runtime
- now: get current time as of engine (`datetime.now` when running in realtime)
- loop: get the event loop for the engine

... and some optional simulators for backtesting.

- slippage
- transactionCost

## 2.3 Exchanges

An exchange instance inherits from two base class, a `MarketData` class which implements data streaming methods, and an `OrderEntry` class which implements order entry methods.

### 2.3.1 Market Data Class

```
class _MarketData(metaclass=ABCMeta):
    '''internal only class to represent the streaming-source
    side of a data source'''

    async def instruments(self):
        '''get list of available instruments'''

    def subscribe(self, instrument):
        '''subscribe to market data for a given instrument'''

    async def tick(self):
        '''return data from exchange'''
```

### 2.3.2 Order Entry Class

```
class _OrderEntry(metaclass=ABCMeta):
    '''internal only class to represent the rest-sink
    side of a data source'''

    def accounts(self) -> List:
        '''get accounts from source'''

    async def newOrder(self, order: Order):
        '''submit a new order to the exchange. should set the given order's `id`␣
↪field to exchange-assigned id

        For MarketData-only, can just return None
        '''

    async def cancelOrder(self, order: Order):
        '''cancel a previously submitted order to the exchange.

        For MarketData-only, can just return None
        '''
```

### 2.3.3 Exchange Class

```
class Exchange(_MarketData, _OrderEntry):
    '''Generic representation of an exchange. There are two primary functionalities␣
↪of an exchange.

    Market Data Source:
        exchanges can stream data to the engine
```

(continues on next page)

(continued from previous page)

```
    Order Entry Sink:
        exchanges can be queried for data, or send data
    '''
    @abstractmethod
    async def connect(self):
        '''connect to exchange. should be asynchronous.

        For OrderEntry-only, can just return None
        '''
```

### 2.3.4 Extending

Writing a custom exchange is very easy, you just need to implement the market data interface, the order entry interface, or both. Here is a simple example of implementing a market data exchange on top of a CSV File, with support for simulated order entry by accepting any trade submitted at the price asked for:

```
import csv
from typing import List
from aat.config import EventType, InstrumentType, Side
from aat.core import ExchangeType, Event, Instrument, Trade, Order
from aat.exchange import Exchange


class CSV(Exchange):
    '''CSV File Exchange'''

    def __init__(self, trading_type, verbose, filename: str):
        super().__init__(ExchangeType('csv-{}'.format(filename)))
        self._trading_type = trading_type
        self._verbose = verbose
        self._filename = filename
        self._data: List[Trade] = []
        self._order_id = 0

    async def instruments(self):
        '''get list of available instruments'''
        return list(set(_.instrument for _ in self._data))

    async def connect(self):
        with open(self._filename) as csvfile:
            self._reader = csv.DictReader(csvfile, delimiter=',')

            for row in self._reader:
                self._data.append(Trade(volume=float(row['volume']),
                                        price=float(row['close']),
                                        maker_orders=[],
                                        taker_order=Order(volume=float(row['volume']),
                                                          price=float(row['close']),
                                                          side=Side.BUY,
                                                          exchange=self.exchange(),
                                                          instrument=Instrument(
                                                              row['symbol'].split('-
↪')[0],
                                                              InstrumentType(row[
↪'symbol'].split('-')[1].upper())
```

(continues on next page)

(continued from previous page)

```
                                        )
                )
                ))

    async def tick(self):
        for item in self._data:
            yield Event(EventType.TRADE, item)

    async def newOrder(self, order: Order):
        if self._trading_type == TradingType.LIVE:
            raise NotImplementedError("Live OE not available for CSV")

        order.id = self._order_id
        self._order_id += 1
        self._queued_orders.append(order)
        return order
```

### 2.3.5 Synthetic Exchange

We provide a sythetic exchange for testing. This exchange produces a variety of equity instruments, and simulates a complete exchange. This exchange runs on the `aat`'s `OrderBook` instance, which supports the following order types:

- Market orders
- Limit orders
- Stop orders

and order flags:

- Fill or kill
- All or none
- Immediate or cancel

The `OrderBook` api is as follows:

```
class OrderBook(object):
    '''A limit order book.

    Supports the following order types:
        - [x] market
            - [x] executes the entire volume
            - [ ] if notional specified, will execute (price*volume) worth (e.g.␣
↪relies on total price, not volume)

            Flags:
                - [x] no flag
                - [x] fill-or-kill: entire order must fill against current book,␣
↪otherwise nothing fills
                - [x] all-or-none: entire order must fill against 1 order, otherwise␣
↪nothing fills
                - [x] immediate-or-cancel: same as fill or kill

        - [x] limit
            - [x] either puts on book or crosses spread, by default puts remainder on␣
↪book
```

(continues on next page)

(continued from previous page)

```
            Flags:
                - [x] no flag
                - [x] fill-or-kill: entire order must fill against current book,␣
↪otherwise cancelled
                - [x] all-or-none: entire order must fill against 1 order, otherwise␣
↪cancelled
                - [x] immediate-or-cancel: whenever this order executes, fill␣
↪whatever fills and cancel remaining

        - [x] stop-market
            - 0 volume order, but when crosses triggers the submission of a market␣
↪order
        - [x] stop-limit
            - 0 volume order, but when crosses triggers the submission of a market␣
↪order

    Supports the following order flags:
        - [x] no flag
        - [x] fill-or-kill
        - [x] all-or-none
        - [x] immediate-or-cancel

    Args:
        instrument (Instrument): the instrument for the book
        exchange_name (str): name of the exchange
        callback (Function): callback on events
    '''
    def add(self, order):
        '''add a new order to the order book, potentially triggering events:
            EventType.TRADE: if this order crosses the book and fills orders
            EventType.FILL: if this order crosses the book and fills orders
            EventType.CHANGE: if this order crosses the book and partially fills␣
↪orders
        Args:
            order (Data): order to submit to orderbook
        '''

    def change(self, order):
        '''modify an order on the order book, potentially triggering events:
            EventType.CHANGE: the change event for this
        Args:
            order (Data): order to submit to orderbook
        '''

    def cancel(self, order):
        '''remove an order from the order book, potentially triggering events:
            EventType.CANCEL: the cancel event for this
        Args:
            order (Data): order to submit to orderbook
        '''

    def find(self, order):
        '''find an order in the order book
        Args:
            order (Data): order to find in orderbook
        '''
```

(continues on next page)

(continued from previous page)

```
    def topOfBook(self):
        '''return top of both sides

        Args:

        Returns:
            value (dict): returns {BUY: tuple, SELL: tuple}
        '''

    def spread(self):
        '''return the spread

        Args:

        Returns:
            value (float): spread between bid and ask
        '''

    def level(self, level: int = 0, price: float = None):
        '''return book level

        Args:
            level (int): depth of book to return
            price (float): price level to look for
        Returns:
            value (tuple): returns ask or bid if Side specified, otherwise ask,bid
        '''

    def levels(self, levels=0):
        '''return book levels starting at top

        Args:
            levels (int): number of levels to return
        Returns:
            value (dict of list): returns {"ask": [levels in order], "bid": [levels␣
↪in order]} for `levels` number of levels
        '''
```

We can also run the `SyntheticExchange` as a service behind websockets to serve as a nice sandbox for testing strategies, building visualizations, etc. To do so, we can run the `aat-synthetic-server` command.

# CHAPTER 3

## Setting up and running

`aat` is setup to run off a configuration file. In this file, we specify some global parameters such as the `TradingType`, as well as configure the `Strategy` and `Exchange` instances.

Let us consider the simple example of the `BuyAndHold` strategy provided above, configured to run in `backtest` mode against the `SyntheticExchange` provided above. Such a configuration file would look like:

```
 > cat myconfig.cfg
[general]
verbose=0
trading_type=backtest

[exchange]
exchanges=
    aat.exchange:SyntheticExchange

[strategy]
strategies =
    aat.strategy.sample:BuyAndHoldStrategy
```

We can run this configuration by running: `aat --config myconfig.cfg`

We can also run via CLI:

```
usage: __main__.py [-h] [--config CONFIG] [--verbose]
                   [--trading_type {live,simulation,sandbox,backtest}]
                   [--strategies STRATEGIES [STRATEGIES ...]]
                   [--exchanges EXCHANGES [EXCHANGES ...]]

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG       Config file
  --verbose             Run in verbose mode
  --trading_type {live,simulation,sandbox,backtest}
                        Trading Type in ("live", "sandbox", "simulation",
                        "backtest")
```

(continues on next page)

(continued from previous page)

```
  --strategies STRATEGIES [STRATEGIES ...]
                        Strategies to run in form
                        <path.to.module:Class,args,for,strat>
  --exchanges EXCHANGES [EXCHANGES ...]
                        Exchanges to run on
```

## 3.1 Trading Type

There are several values for the `TradingType` field:

- `live` - live trading against the exchange
- `simulation` - live trading against the exchange, but with order entry disabled
- `sandbox` - live trading against the exchange's sandbox or paper trading instance
- `backtest` - offline trading against historical OHLCV data

To test our strategy in any mode, we may need to setup exchange-specific keys to get historical data, stream market data, and make new orders.

## 3.2 Strategies and Exchanges

We can run any number of strategies against any number of exchanges, including custom user-defined strategies and exchanges not implemented in the core `aat` repository. `aat` will multiplex the event streams and your strategies control which instruments they trade against which exchanges.

| Exchange | Market Data | Order Entry | TradingTypes | Asset Classes | |—|—|—|—|—| | Synthetic | Yes | Yes | Simulation,Backtest | Equity | | IEX | Yes | Fake | Live, Simulation, Sandbox, Backtest | Equity | | InteractiveBrokers | In Progress | Yes | Live, Simulation, Sandbox | Equity, Option, Future, Commodities, Spreads, Pairs | | TD Ameritrade | In Progress | In Progress | Equity, Option | | Alpaca | In Progress | In Progress | | | Coinbase | In Progress | In Progress | | | Gemini | In Progress | In Progress | | | Coinbase | In Progress | In Progress | | | ccxt | In Progress | In Progress | |

# CHAPTER 4

TODO below here are sections that still need to be documented

## 4.1 Core Data Structures

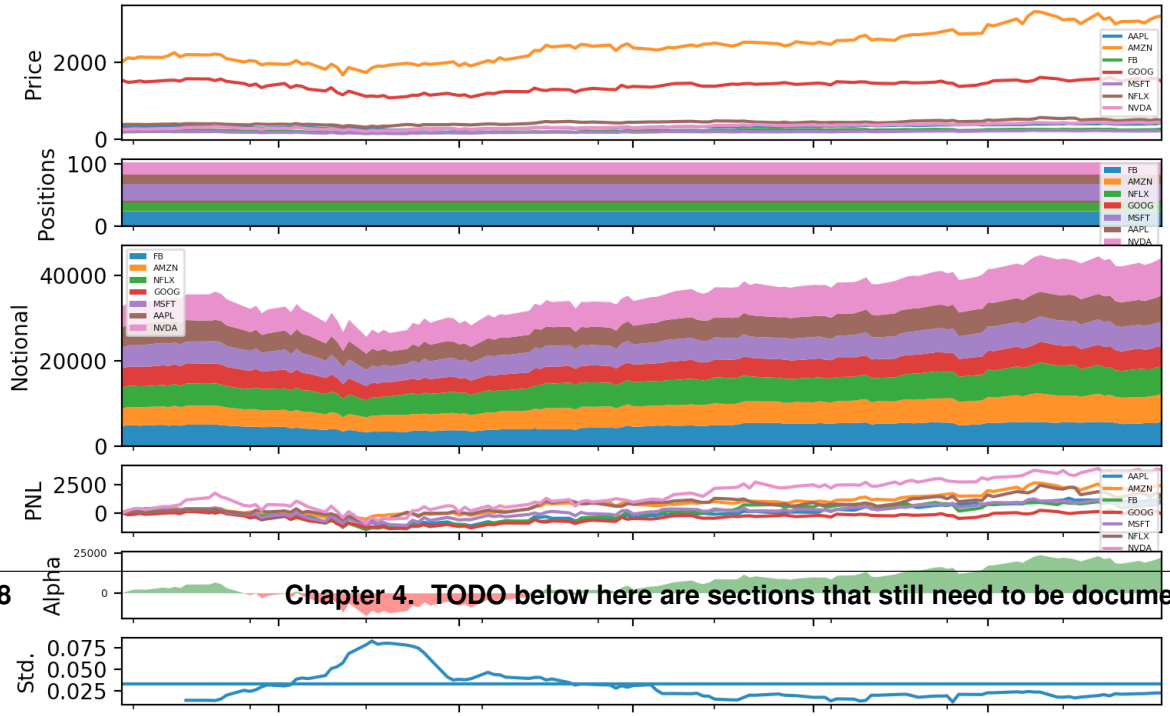
### 4.1.1 Enums

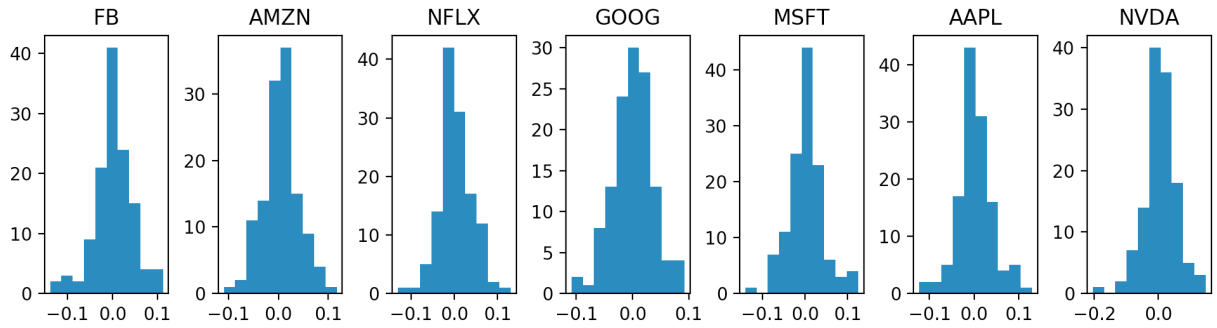
### 4.1.2 Models

### 4.1.3 Instruments

## 4.2 Other Features

### 4.2.1 Trade/Portfolio Analysis






### 4.2.2 API Access

### 4.2.3 Risk Management

### 4.2.4 Execution# API Documentation

Logging package for Python. Based on PEP 282 and comments thereto in comp.lang.python.



To use, simply 'import logging' and log away!

**class** logging.**BufferingFormatter**(*linefmt=None*)
Bases: object

A formatter suitable for formatting a number of records.

**format**(*records*)
Format the specified records and return the result as a string.

**formatFooter**(*records*)
Return the footer string for the specified records.

**formatHeader**(*records*)
Return the header string for the specified records.

**class** logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)
Bases: *logging.StreamHandler*

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

**close**()
Closes the stream.

**emit**(*record*)
Emit a record.

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

**class** logging.**Filter**(*name=''*)
Bases: object

Filter instances are used to perform arbitrary filtering of LogRecords.

Loggers and Handlers can optionally use Filter instances to filter records as desired. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with "A.B"

will allow events logged by loggers “A.B”, “A.B.C”, “A.B.C.D”, “A.B.D” etc. but not “A.BB”, “B.A.B” etc. If initialized with the empty string, all events are passed.

**filter**(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

**class** logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Bases: object

Formatter instances are used to convert a LogRecord to text.

Formatters need to know how a LogRecord is constructed. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the the style-dependent default value, “%(message)s”, “{message}”, or “${message}”, is used.

The Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - e.g. the default value mentioned above makes use of the fact that the user’s message and arguments are pre-formatted into a LogRecord’s message attribute. Currently, the useful attributes in a LogRecord are described by:

%(name)s Name of the logger (logging channel) %(levelno)s Numeric logging level for the message (DEBUG, INFO,

WARNING, ERROR, CRITICAL)

**%(levelname)s Text logging level for the message (“DEBUG”, “INFO”,** “WARNING”, “ERROR”, “CRITICAL”)

**%(pathname)s Full pathname of the source file where the logging** call was issued (if available)

%(filename)s Filename portion of pathname %(module)s Module (name portion of filename) %(lineno)d Source line number where the logging call was issued

(if available)

%(funcName)s Function name %(created)f Time when the LogRecord was created (time.time()

return value)

%(asctime)s Textual time when the LogRecord was created %(msecs)d Millisecond portion of the creation time %(relativeCreated)d Time in milliseconds when the LogRecord was created,

relative to the time the logging module was loaded (typically at application startup time)

%(thread)d Thread ID (if available) %(threadName)s Thread name (if available) %(process)d Process ID (if available) %(message)s The result of record.getMessage(), computed just as

the record is emitted

**converter**()

**localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,**
tm_sec,tm_wday,tm_yday,tm_isdst)

Convert seconds since the Epoch to a time tuple expressing local time. When ‘seconds’ is not passed in, convert the current time instead.

**default_msec_format = '%s,%03d'**

**default_time_format = '%Y-%m-%d %H:%M:%S'**

**format**(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

**formatException**(*ei*)
Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

**formatMessage**(*record*)

**formatStack**(*stack_info*)
This method is provided as an extension point for specialized formatting of stack information.

The input data is a string as returned from a call to `traceback.print_stack()`, but with the last trailing newline removed.

The base implementation just returns the value passed in.

**formatTime**(*record*, *datefmt=None*)
Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, an ISO8601-like (or RFC 3339-like) format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

**usesTime**()
Check if the format uses the creation time of the record.

**class** logging.**Handler**(*level=0*)
Bases: logging.Filterer

Handler instances dispatch logging events to specific destinations.

The base handler class. Acts as a placeholder which defines the Handler interface. Handlers can optionally use Formatter instances to format records as desired. By default, no formatter is specified; in this case, the 'raw' message as determined by record.message is logged.

**acquire**()
Acquire the I/O thread lock.

**close**()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

**createLock**()
Acquire a thread lock for serializing access to the underlying I/O.

**emit**(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

**flush**()
Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

**format**(*record*)
Format the specified record.

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

**get_name**()

**handle**(*record*)
Conditionally emit the specified logging record.

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

**handleError**(*record*)
Handle errors which occur during an emit() call.

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

**name**

**release**()
Release the I/O thread lock.

**setFormatter**(*fmt*)
Set the formatter for this handler.

**setLevel**(*level*)
Set the logging level of this handler. level must be an int or a str.

**set_name**(*name*)

**class** logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*, ***kwargs*)
Bases: object

A LogRecord instance represents an event being logged.

LogRecord instances are created every time something is logged. They contain all the information pertinent to the event being logged. The main information passed in is in msg and args, which are combined using str(msg) % args to create the message field of the record. The record also includes information such as when the record was created, the source line where the logging call was made, and any exception information to be logged.

**getMessage**()
Return the message for this LogRecord.

Return the message for this LogRecord after merging any user-supplied arguments with the message.

**class** logging.**Logger**(*name*, *level=0*)
Bases: logging.Filterer

Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g.

an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

**addHandler**(*hdlr*)
Add the specified handler to this logger.

**callHandlers**(*record*)
Pass a record to all relevant handlers.

Loop through all handlers for this logger and its parents in the logger hierarchy. If no handler was found, output a one-off error message to sys.stderr. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger whose handlers are called.

**critical**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

**debug**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

**error**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

**exception**(*msg*, **args*, *exc_info=True*, ***kwargs*)
Convenience method for logging an ERROR with exception information.

**fatal**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

**findCaller**(*stack_info=False*)
Find the stack frame of the caller so that we can note the source file name, line number and function name.

**getChild**(*suffix*)
Get a logger which is a descendant to this one.

This is a convenience method, such that

logging.getLogger('abc').getChild('def.ghi')

is the same as

logging.getLogger('abc.def.ghi')

It's useful, for example, when the parent logger is named using __name__ rather than a literal string.

**getEffectiveLevel**()
Get the effective level for this logger.

Loop through this logger and its parents in the logger hierarchy, looking for a non-zero logging level. Return the first one found.

**handle**(*record*)
Call the handlers for the specified record.

This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied.

**hasHandlers**()
See if this logger has any handlers configured.

Loop through all handlers for this logger and its parents in the logger hierarchy. Return True if a handler was found, else False. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger which is checked for the existence of handlers.

**info**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1)

**isEnabledFor**(*level*)
Is this logger enabled for level 'level'?

**log**(*level*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with the integer severity 'level'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.log(level, "We have a %s", "mysterious problem", exc_info=1)

**makeRecord**(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
A factory method which can be overridden in subclasses to create specialized LogRecords.

**manager = <logging.Manager object>**

**removeHandler**(*hdlr*)
Remove the specified handler from this logger.

**root = <RootLogger root (WARNING)>**

**setLevel**(*level*)
Set the logging level of this logger. level must be an int or a str.

**warn**(*msg*, **args*, ***kwargs*)

**warning**(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

**class** logging.**LoggerAdapter**(*logger*, *extra*)
Bases: object

An adapter for loggers which makes it easier to specify contextual information in logging output.

**critical**(*msg*, **args*, ***kwargs*)
Delegate a critical call to the underlying logger.

**debug**(*msg*, **args*, ***kwargs*)
Delegate a debug call to the underlying logger.

**error**(*msg*, **args*, ***kwargs*)
Delegate an error call to the underlying logger.

**exception**(*msg*, **args*, *exc_info=True*, ***kwargs*)
Delegate an exception call to the underlying logger.

**getEffectiveLevel**()
Get the effective level for the underlying logger.

**hasHandlers**()
See if the underlying logger has any handlers.

**info**(*msg*, **args*, ***kwargs*)
Delegate an info call to the underlying logger.

**isEnabledFor**(*level*)
Is this logger enabled for level 'level'?

**log**(*level*, *msg*, **args*, ***kwargs*)
Delegate a log call to the underlying logger, after adding contextual information from this adapter instance.

**manager**

**name**

**process**(*msg*, *kwargs*)
Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

**setLevel**(*level*)
Set the specified level on the underlying logger.

**warn**(*msg*, **args*, ***kwargs*)

**warning**(*msg*, **args*, ***kwargs*)
Delegate a warning call to the underlying logger.

**class** logging.**NullHandler**(*level=0*)
Bases: *logging.Handler*

This handler does nothing. It's intended to be used to avoid the "No handlers could be found for logger XXX" one-off warning. This is important for library code, which may contain code to log events. If a user of the library does not configure logging, the one-off warning might be produced; to avoid this, the library developer simply needs to instantiate a NullHandler and add it to the top-level logger of the library module or package.

**createLock**()
Acquire a thread lock for serializing access to the underlying I/O.

**emit**(*record*)
Stub.

**handle**(*record*)
Stub.

**class** logging.**StreamHandler**(*stream=None*)
Bases: *logging.Handler*

A handler class which writes logging records, appropriately formatted, to a stream. Note that this class does not close the stream, as sys.stdout or sys.stderr may be used.

**emit**(*record*)
Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

**flush**()
Flushes the stream.

**setStream**(*stream*)
Sets the StreamHandler's stream to the specified value, if it is different.

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

**terminator = '\n'**

logging.**addLevelName**(*level*, *levelName*)
Associate 'levelName' with 'level'.

This is used when converting levels to text during message formatting.

logging.**basicConfig**(***kwargs*)
Do basic configuration for the logging system.

This function does nothing if the root logger already has handlers configured. It is a convenience method intended for use by simple scripts to do one-shot configuration of the logging package.

The default behaviour is to create a StreamHandler which writes to sys.stderr, set a formatter using the BASIC_FORMAT format string, and add the handler to the root logger.

A number of optional keyword arguments may be specified, which can alter the default behaviour.

**filename Specifies that a FileHandler be created, using the specified** filename, rather than a StreamHandler.

**filemode Specifies the mode to open the file, if filename is specified** (if filemode is unspecified, it defaults to 'a').

format Use the specified format string for the handler. datefmt Use the specified date/time format. style If a format string is specified, use this to specify the

type of format string (possible values '%', '{', '$', for %-formatting, `str.format()` and `string.Template` - defaults to '%').

level Set the root logger level to the specified level. stream Use the specified stream to initialize the StreamHandler. Note

that this argument is incompatible with 'filename' - if both are present, 'stream' is ignored.

**handlers If specified, this should be an iterable of already created** handlers, which will be added to the root handler. Any handler in the list which does not have a formatter assigned will be assigned the formatter created in this function.

Note that you could specify a stream created using open(filename, mode) rather than passing the filename and mode in. However, it should be remembered that StreamHandler does not close its stream (since it may be using sys.stdout or sys.stderr), whereas FileHandler closes its stream when the handler is closed.

Changed in version 3.2: Added the `style` parameter.

Changed in version 3.3: Added the `handlers` parameter. A `ValueError` is now thrown for incompatible arguments (e.g. `handlers` specified together with `filename`/`filemode`, or `filename`/`filemode` specified together with `stream`, or `handlers` specified together with `stream`.

logging.**captureWarnings**(*capture*)
If capture is true, redirect all warnings to the logging package. If capture is False, ensure that warnings are not redirected to logging but to their original destinations.

logging.**critical**(*msg*, **args*, ***kwargs*)
Log a message with severity 'CRITICAL' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**debug**(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**disable**(*level=50*)
Disable all logging calls of severity 'level' and below.

logging.**error**(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**exception**(*msg*, **args*, *exc_info=True*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger, with exception information. If the logger has no handlers, basicConfig() is called to add a console handler with a pre-defined format.

logging.**fatal**(*msg*, **args*, ***kwargs*)
Log a message with severity 'CRITICAL' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**getLevelName**(*level*)
Return the textual representation of logging level 'level'.

If the level is one of the predefined levels (CRITICAL, ERROR, WARNING, INFO, DEBUG) then you get the corresponding string. If you have associated levels with names using addLevelName then the name you have associated with 'level' is returned.

If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

Otherwise, the string "Level %s" % level is returned.

logging.**getLogger**(*name=None*)
Return a logger with the specified name, creating it if necessary.

If no name is specified, return the root logger.

logging.**getLoggerClass**()
Return the class to be used when instantiating a logger.

logging.**info**(*msg*, **args*, ***kwargs*)
Log a message with severity 'INFO' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**log**(*level*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with the integer severity 'level' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**makeLogRecord**(*dict*)
Make a LogRecord whose attributes are defined by the specified dictionary, This function is useful for converting a logging event received over a socket connection (which is sent as a dictionary) into a LogRecord instance.

logging.**setLoggerClass**(*klass*)
Set the class to be used when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__()

logging.**shutdown**(*handlerList=[<weakref at 0x7f145cf189a8; to '_StderrHandler'>, <weakref at 0x7f145a7365e8; to 'NewLineStreamHandlerPY3'>, <weakref at 0x7f145a7366d8; to 'WarningStreamHandler'>, <weakref at 0x7f145a7367c8; to 'StreamHandler'>, <weakref at 0x7f1459ee6868; to 'NullHandler'>, <weakref at 0x7f1459dd44f8; to 'NullHandler'>, <weakref at 0x7f1459a64f48; to 'MemoryHandler'>]*)

Perform any cleanup actions in the logging system (e.g. flushing buffers).

Should be called at application exit.

logging.**warn**(*msg*, **args*, ***kwargs*)

logging.**warning**(*msg*, **args*, ***kwargs*)

Log a message with severity 'WARNING' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

logging.**getLogRecordFactory**()

Return the factory to be used when instantiating a log record.

logging.**setLogRecordFactory**(*factory*)

Set the factory to be used when instantiating a log record.

**Parameters** **factory** – A callable which will be called to instantiate

a log record.

This is an interface to Python's internal parser.

**exception** parser.**ParserError**

Bases: Exception

parser.**STType**

alias of parser.st

parser.**compilest**()

Compiles an ST object into a code object.

parser.**expr**()

Creates an ST object from an expression.

parser.**isexpr**()

Determines if an ST object was created from an expression.

parser.**issuite**()

Determines if an ST object was created from a suite.

parser.**sequence2st**()

Creates an ST object from a tree representation.

parser.**st2list**()

Creates a list-tree representation of an ST.

parser.**st2tuple**()

Creates a tuple-tree representation of an ST.

parser.**suite**()

Creates an ST object from a suite.

parser.**tuple2st**()

Creates an ST object from a tree representation.

# Python Module Index

# Index