# blackblox Documentation

*Release 0.2.0*

S.E. Tanzer

Mar 10, 2019

# CONTENTS

This is the documentation for the nascent BlackBlox.py, which aims to faciliate the creation and exploration of black box models of aribitrary levels of details by automating mass and energy balances and generation of pretty (and not so pretty) figures.

This is currently a private project and very much a work-in-progress.

CHAPTER

# ONE

# CONTENTS

## 1.1 What is blackblox.py?

blackblox is a calculator for “black box” systems, ranging from single unit processes to factories with branching chains of processes.

**# NOTE: This introductory document still must be updated with the latest feature set**

### 1.1.1 Unit Processes

Unit processes are the smallest”block” in blackblox. Each unit process has a set of inflows and outflows and a set of specified relationships between the process flows. Then, given a quantity for one inflow or outflow, the quantities of the remaining inflows and outflows can be calculated.

#### Defining unit processes

A unit process is defined by two tables:

#### Calculations Table

The first specifies the relationships between inflows and outflows, with each row listing two substances; whether each substance is an inflow, an outflow, or a flow internal to the unit process; the type of calculation that would generate the quantity of the second substance if the quantity of the first substance is known, and the name of the variable (if any) used in that calculation.

The calculation types must be those available in the program’s calculator library. The substance variable names are user specified. It is also possible to define special substance names (e.g. “fuel”) to allow the substance to be defined in the variables table (e.g. “fuel type”) and also have properties defined elsewhere (e.g. HHV and combustion emissions)

e.g.

There is also a feature

#### Variables Table

The second provides the values of the variables named in the calculations table. Separating the values into their own table allows for the same unit process to be run in different configurations(e.g. different efficiencies or fuel types).

e.g.

#### Balancing unit processes

Balancing a unit process calculates the quantity of all inflows and outflows of that unit process. To balance a unit process, the following arguments must be provided:

- the quantity of one inflow or outflow
- the name of that inflow or outflow substance
- whether the substance is an inflow or outflow
- the name of the configuration scenario to use from the variables table.

All arguments besides the quantity can be optional, if default values have been specified for the unit process.

After calculating all flows based on the user input, blackblox then checks for whether the mass and energy flows are balanced, and either raises an exception or, by default, creates an 'UKNOWN' balancer flow where needed.

blackblox assumes that flows are mass by defautl. It decides if a flow is an energy flow based on whether it begins or end with an energy-signifier string (such as 'energy', 'heat', or 'electricity'), which the user can specify.

N.B. When processing fuel combustion, blackblox writes an "energy in [fuelname]" flow to the inflows dictionary, to balance the energy of combustion.

#### Unit Table Library

A library table with a list to the locations of all the unit processes available will allow you to not have to enter in the data every time the unit process function is called.

e.g.

### 1.1.2 Process Chain

A process chain is a linear collection of one or more connected unit processes, where an outflow of a preceding unit process is an inflow of the following unit process.

#### Defining a chain

A process chain is defined by a table with a list of unit processes with an inflow and outflow to each, where the outflow of a unit process must be the inflow into the next unit process. When the process chain is first used, an initializalion process creates each of the unit processes, if they do not already exist, and verifies that the inflows and outflows specified in the chain table exist for the corresponding unit processes.

e.g.

#### Balancing a chain

Balancing a chain calculates the quantity of all inflows and outflows of each unit process in the chain, either from first inflow to last outflow or from last outflow to first inflow. To balance a chain, the following arguments must be provided:

- the quantity of one inflow or outflow somwhere in the chain
- the name of that inflow or outflow substance
- the name of the configuration scenario to use from the variables table.
- the name of the unit process in the chain, if the specified flow is not an iflow into the first process or an outflow of the last proces.

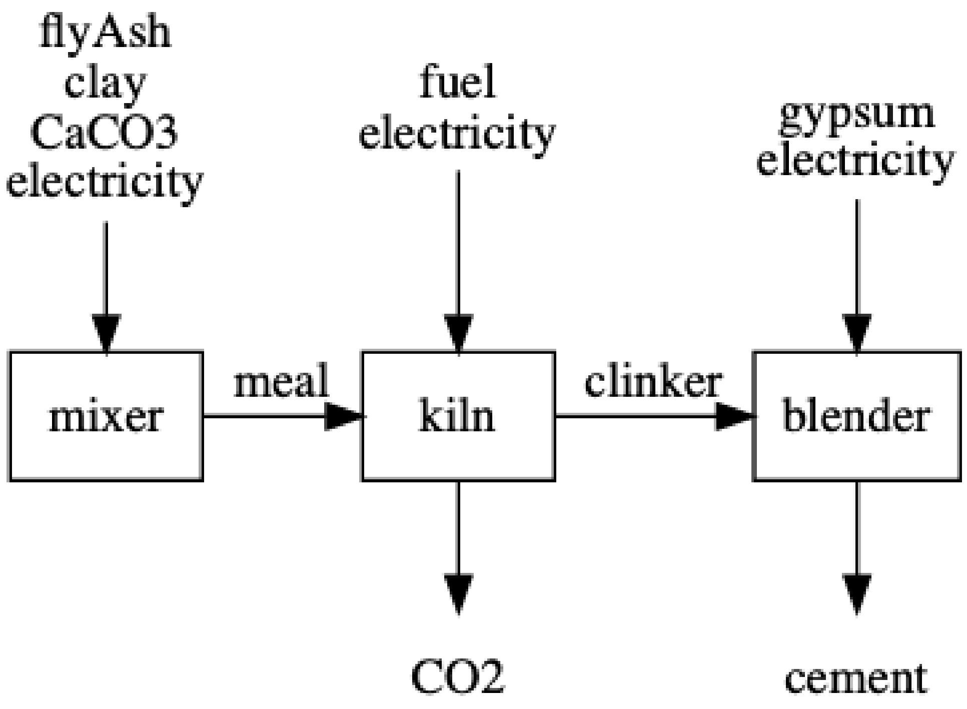
All arguments besides the quantity can be optional, if default values have been specified for the process chain.

Balancing a chain returns a dictionary with both the calculated inflows and outflows for each unit process, as well as the the overall inflows and outflows.

#### Generating a chain diagram

After a chain has been defined, a process flow diagram of the chain can be generated.

E.g.



### 1.1.3 Factory

A factory is a collection of one or more connected process chains, where the inflow of outflow of any unit process within any chain can be the inflow or outflow of any other chain. A factory has a single main chain, and zero or more auxiliary chains. By specifying an input or output quantity to the main chain, it is possible to calculate the inflows and outflows of all processes within the chain.

#### Defining a factory

A factory is defined by two tables: one that lists the chains (and their file locations) and another that lists the connections between the chains. Factory data can be in seperate spreadsheets or in a single excel workbook.

#### Factory chains table

This table specifies the location of the proces chain data used in the factory, as well as the primary product and whether that product is an inflow or outflow of the chain. The first chain of the chain list is assumed to be the main product chain, but otherwise it doesn't matter. The chains can either be in the same file, if an excel workbook, or in a seperate file.

e.g.

e.g.

It is also possible to specify recycling flows in the connections table, if a Destination Unit is also provided. (The optional destination unit can be used to send flows to intermediate unit processes in chains.) Currently two types of recycling are usable:

- *1-to-1 replacement*: allows for a recycle flow to replace a flow in another unit process, given that is a 1-to-1 correspondent and does not affect other flows in unit process
- *energy replacing fuel*: allows for recycled energy to replace energy generated from a fuel in a unit process that has a combustion calculation. The emission outflows (e.g. CO2) and inflow of oxygen are also recalculated.

It is possible to specify a purge fraction of the initial flow, as well as a maximum percentage of the existing flow that the recycled flow can replace. There can be recycling flow leftover, which is treated as a normal output. If there is insufficient recycling flow to fully replace the existing flow, it is only partially replaced. A single flow can be recycled into multiple units, by specifying each connection as a new line in the table, but only up to the original quantity of the recycling flow.

### Balancing a factory

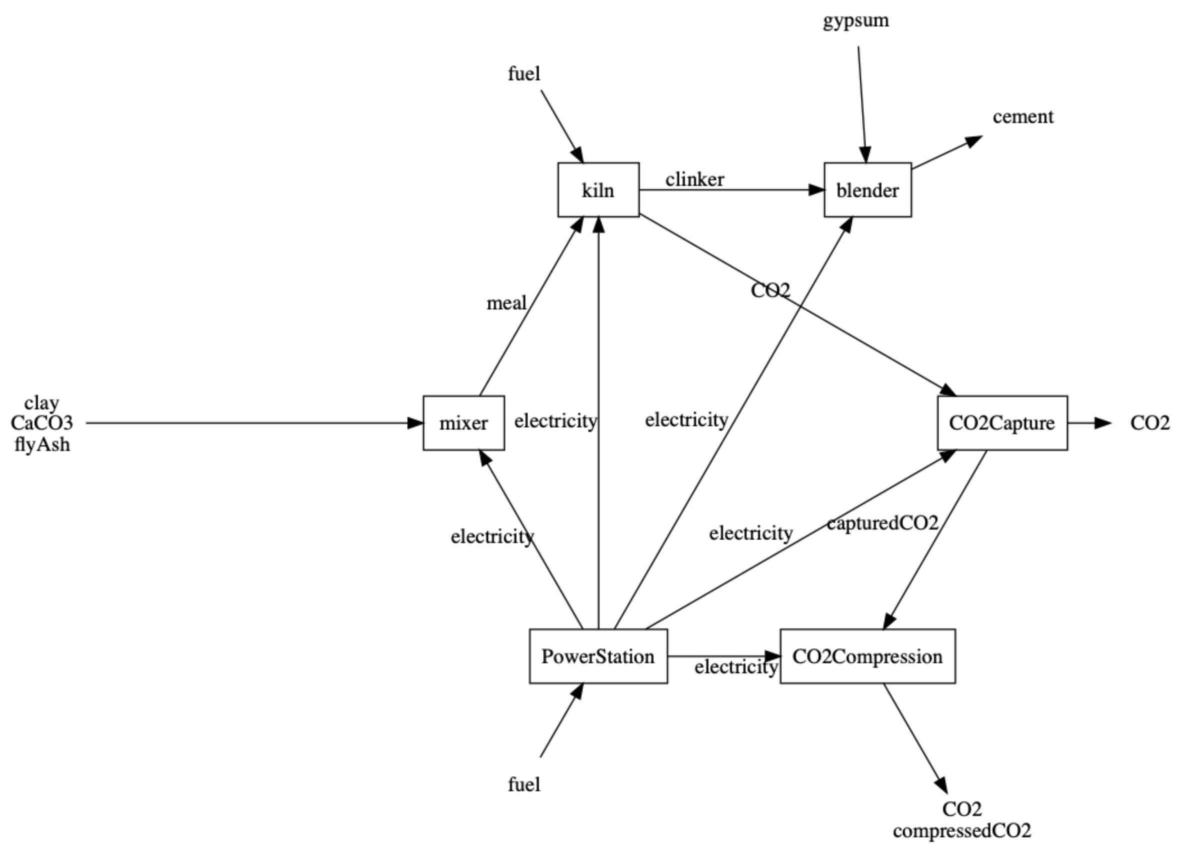
A factory is balanced on a single quantity of the main product and using a single scenario of variables for the whole factory. Balancing a factory returns a nested dictionary with the inflows and outflows of the factory, every chain, and every unit process. Optionally, the data can be outputted to an Excel workbook with sheets for:

- Total factory inflows and outflows
- A matrix for all inflow from every unit process
- A matrix for all outflows from every unit process
- A list of all internal flows/connections, both within chains and
- Inflows for each chain unit process and chain totals (per chain)
- Outflows for each chain unit process and chain totals (per chain)

By default, the data is divided into mass and energy flows, if energy flow name signifiers are provided, with totals for both mass and energy flows.

### Generating a factory diagram

After a factory has been defined, a process flow diagram of the factory can be generated. However, due to the limitations of the diagram rendering software, they are unlikely to be pretty.



E.g.

### 1.1.4 Industries

The largest block is currently an "Industry", which is a collection of one or more factories that produce one or more products.

#### Defining an Industry

An industry is defined by a factory list table, which has the name of each factory and location of its data.

E.g.

#### Balancing an Industry

An industry can be balanced by specifying the output product quantity for each factory, and the scenario of variables for each factory. It is possible to specify each of these absolutely for each factory, or relative to the total output of each product. It is possible to mix absolute and relative specifications in an industry, but it must be the same for each product. It is also possible to batch-balance the industries on the same output quantity but using different scenarios of variable.

E.g. Absolute Product and Scenarios

E.g. Relative products and scenarios

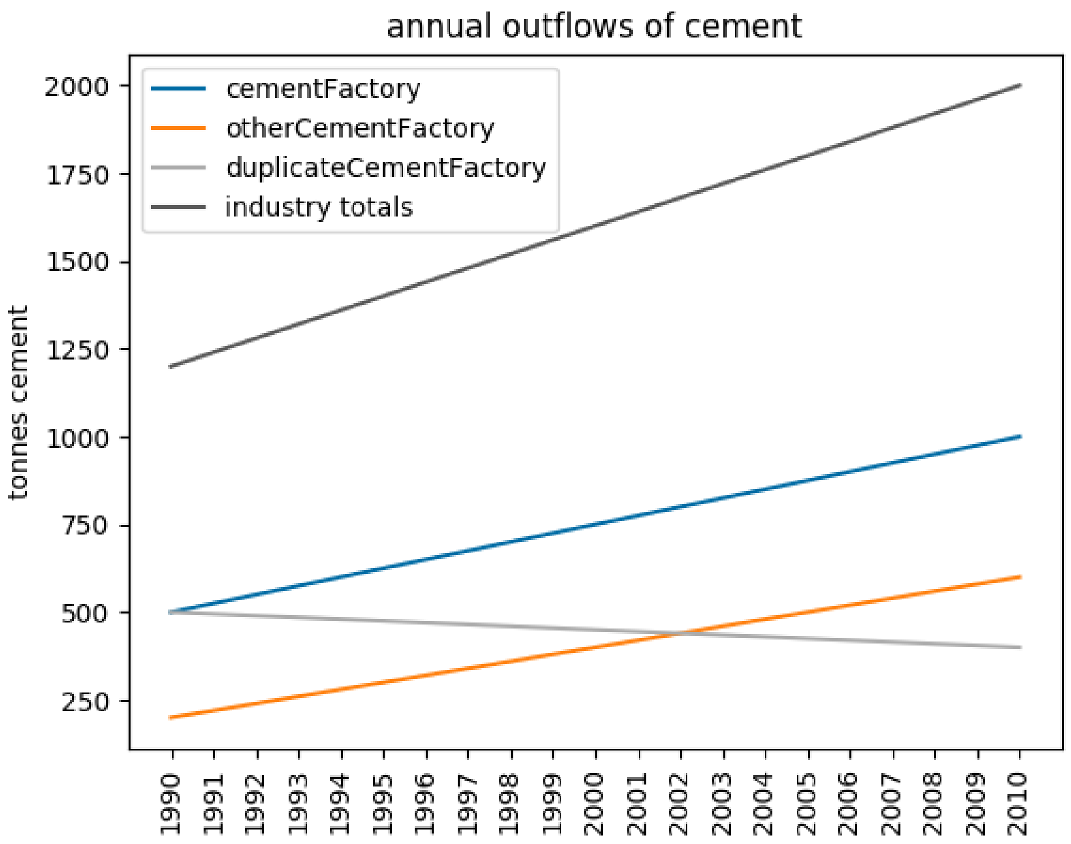
Balancing an industry returns a dictionary with the inputs and outputs of every factory and sub-block. It is also possible to output the industry data to a set of Excel workbooks, one for each factory (as described above), and one for the industry as a whole, with matricies of overall inputs and outputs by factory and for the industry in total.

#### Evolving an Industry

Evolving an industry allows for the specification of a start timestep industry state and an end timestep industry state, using the same format for balancing an industry. blackblox can then calculate both annual and cumulative inflows and outflows, assuming a linear change between start and industry state. It is possible for one state to have a different set of factories than the other, as long as all the factories were listed in the industry factory list table.

Evolving an industry outputs the same Excel files as does balancing an industry —one set for each the start and end state—, as well as a file with the cumulative and annual inflows and outflows listed by year, both for the industry as a whole, and for all factories.

You can also export a graph of the annual outputs of a given flow, e.g.



### 1.1.5 Forthcoming Features

- Specifying evolution growth rates rather than absolute values
- Multi-step evolution of industries
- Comparison of industries (static and evolution)
- More and prettier graphics
- Cleaner code with beter documentation

### 1.1.6 Current limitations

- No more than two “must be known” quantities can be specified in the calculations
- Recycling flows cannot affect other inflows/outflows in a chain

  - excepting emissions and oxygen from combustion when replacing fuel with recycled energy

- Chains can only balance if the specified flow is a substance name that is uniquely an input or uniquely an output of the chain.
- Industries only can evolve linearly.
- Mass flows cannot carry energy or other properties - this would require treating flows as objects, and I'm not sure I'm going to do that
  - Therefore energy flows will not balance
  - Though fuels are recognized as containing energy, if specified in the fuels lookup dataframe

## 1.2 The Unit Process Module

Unit process class

This module contains the Unit Process class, which is the smallest, and fundamental, block used in BlackBlox.py.

Each unit process has inflows and outflows, whose abstract relationships are specified in a relationships table. The numeric values for the variables used in those relationships are specified in a seperate table, to allow for multipe scenarios to be used.

The Unit Process class has a single function, which is to balance the inflows and outflows given one quantity of an inflow or outflow. Balancing the unit process requires that the relationships table is complete.

Module Outline:

- import statements and logger
- module variable: df_units_library (dataframe)
- **class: unit process**
  - class function: Balance
  - class function: recycle_1to1
  - class function: recycle_energy_replacing_fuel

### 1.2.1 UnitProcess Class

**class** unitprocess.**UnitProcess**(*name*, *var_df=False*, *calc_df=False*, *units_df=df_unit_library*)

Unit processes have inflows and outflows with defined relationships.

The relationships of the unit process flows must be defined so that

**Parameters**

- **name** (*str*) – Unique name for the process
- **var_df** (*str/dataframe/bool*) – Optional. Dataframe or filepath tof tabular data of the variable values to use when balancing the unit process. If False, __init__ uses df_unit_library to fetch data. (Defaults to False)
- **calc_df** (*str/dataframe/bool*) – Optional. Dataframe or filepath to tabular data the relationships between the flows within the unit process, with each relationship having no more than a single variable. If False, __init__ uses df_unit_library to fetch data. The relationships must be sufficiently specified so that so that there are zero degrees of freedom. (Defaults to False)
- **units_df** (*dataframe*) – Unit process library data frame (Defaults to df_unit_library)

**name**
Name of process

**Type** str

**var_df**
Dataframe of relationship variable values indexed by scenario name.

**Type** dataframe

**calc_df**
Dataframe of relationships between unit process flows.

**Type** dataframe

**default_product**
The primary "product" flow of the unit process. Derived from units_df.

**Type** str

**default_io**
Whether the primary product is an inflow or an outflow. Derived from units_df.

**Type** str

**inflows**
List of inflows to the unit process. Derived from calc_df.

**Type** set[str]

**outflows**
List of outflows to the unit process. Derived from calc_df.

**Type** set[str]

### 1.2.2 UnitProcess.balance()

`unitprocess.UnitProcess.`**`balance`**(*self*, *qty*, *product=False*, *i_o=False*, *scenario=dat.default_scenario*, *energy_flows=dat.energy_flows*, *balance_energy=True*, *raise_imbalance=False*)

performs a mass (and optionally energy) balance on the unit process.

Calculates all inflows and outflows, using the specified variable values and relationship dataframe. Will only work if there are zero degrees of freedom in the specified relationships. Currently, only checks mass flows to see if they are properly balanced.

**Note:** If the inflow mass and outflow mass are imbalanced an error will be raised and/or a "UNKNOWN MASS" or "UNKNOWN ENERGY " flow will be added to the offending flow dictionary with the imbalance quantity.

**Parameters**

- **qty** (*float*) – The quantity of the specified flow.
- **product** (*str/bool*) – the inflow or outflow on which to balance the calculations. If False, uses the default product. (Defaults to False)
- **i_o** (*str*) – 'i' or 'o', depending on whether the specified product is an inflow (i) or outflow (o). If False, uses the default product's IO. (Defaults to False)
- **scenario** (*str*) – row index of var_df to use for the variables value, generally corresponding to the name of the scenario. If False, uses the default scenario index specified in dataconfig. (Defaults to False)

- **energy_flows** (*list[str]*) – If any substance name starts with or ends with a string on this list, the substance will not be considered when performing the mass balance. Defaults to energy keyword list in dataconfig.
- **balance_energy** (*bool*) – If true, checks for a balance of the flows with the specified energy prefix/suffix in energy_flows.
- **raise_imbalance** (*bool*) – If True, the process will raise an exception if the inflow and outflow masses and/or energies are unbalanced. If False, will add a "UNKNOWN MASS" or "UNKNOWN ENERGY" substance to the offended inflow or outflow dictionary. Defaults to False.

**Returns** Defaultdict of inflows with substance names as keys and quantities as values Defaultdict of outflows with substance names as keys and quantities as values.

### 1.2.3 UnitProcess.recycle_1to1()

`unitprocess.UnitProcess.`**`recycle_1to1`**(*self*, *original_inflows_dict*, *original_outflows_dict*, *recycled_qty*, *recycle_io*, *recycled_flow*, *replaced_flow*, *max_replace_fraction=1.0*, *scenario='default'*, ***kwargs*)

Inserts a recycle flow that corresponds 1-to-1 with an existing flow

**Parameters**

- **original_inflows_dict** (*defaultdict*) – Dictionary of inflow quantities from the orignal balancing of the unit process
- **original_outflows_dict** (*defaultdict*) – Dictionary of outflow quantities from the orignal balancing of the unit process
- **recycled_qty** (*float*) – quantity of recycled flow
- **recycle_io** (*str*) – "i" if the recycled flow is an inflow or "o" if it is an outflow
- **recycled_flow** (*str*) – name of the recycled flow
- **replaced_flow** (*str*) – name of the flow to be replaced by the recycled flow
- **max_replace_fraction** (*float/none*) – the maximum percentage of the original flow that the recycled flow is allowed to replace

**Returns**

- *dictionary* of rebalanced inflows
- *dictionary* of rebalanced outflows
- *float* of the remaining quantity of the recycle stream

### 1.2.4 UnitProcess.recycle_energy_replacing_fuel()

`unitprocess.UnitProcess.`**`recycle_energy_replacing_fuel`**(*original_inflows_dict, original_outflows_dict, recycled_qty, recycle_io, recycled_flow, replaced_flow, max_replace_fraction=1.0, combustion_eff = dat.combustion_efficiency_var, scenario=dat.default_scenario, emissions_list = ['CO2', 'H2O', 'SO2'], **kwargs*)

replaces fuel use and associated emissions with a recycled energy flow

**Parameters**

- **`original_inflows_dict`** (*`defaultdict`*) – Dictionary of inflow quantities from the orignal balancing of the unit process
- **`original_outflows_dict`** (*`defaultdict`*) – Dictionary of outflow quantities from the orignal balancing of the unit process
- **`recycled_qty`** (*`float`*) – quantity of recycled flow
- **`recycle_io`** (*`str`*) – "i" if the recycled flow is an inflow or "o" if it is an outflow
- **`recycled_flow`** (*`str`*) – name of the recycled flow
- **`replaced_flow`** (*`str`*) – name of the flow to be replaced by the recycled flow
- **`[str/float]`** (*`combustion_eff`*) – The name of the combustion efficiency variable (case sensitive) from the variables table or a float of the combustion_eff (must be between 0 and 1)
- **`scenario`** (*`str`*) – name of the scenario to use (Defaults to dat.default_scenario)
- **`emissions_list`** (*`list[str]`*) – list of emissions to recalculation. O2 is always automatically recalculated. (Defaults to ['CO2', 'H2O', 'SO2'])

**Returns**

- *dictionary* of rebalanced inflows
- *dictionary* of rebalanced outflows
- *float* of the remaining quantity of the recycle stream

### 1.2.5 Unit and Variable Libraries

`unitprocess.`**`df_unit_library`**

dataframe of all unit process names and file locations

This data frame is used by the unit process class to provide the locations of the calculations and variable tables for each unit process. Data locations for each unit process can also be provided invidivually, but if not otherwise specified, the unit process __init__ function will look for the data in this data frame.

The names of the unit process should be index column of the data frame, and there should also be columns for the file paths of the variables table and calculations tables. Columns for sheet names, if the data is within excel sheets, will also be used if provided.

`unitprocess.`**`lookup_var_dict`**

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs

**dict(iterable) -> new dictionary initialized as if via:** d = { } for k, v in iterable:

d[k] = v

**dict(**kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

## 1.3 The Process Chain Module

Product Chain class

This module contains the Product Chain class, which are objects that link together (and can generate) a linear sequence of unit processes.

Product chains can currently be balanced on an inflow of their initial unit process or an outflow of their final unit process. It is also possible to automatically generate diagrams of the flows in a product chain.

Module Outline:

- import statements and logger
- **class: ProductChain**
  - class function: Build
  - class function: Balance
  - class function: Diagram

### 1.3.1 ProductChain Class

**class** processchain.**ProductChain**(*chain_data*, *name='Product Chain'*, *xls_sheet=None*)

Linear linkage of unit processes.

Product chains are a linear sets of unit process linked by their inflows and outflow; the outflow of a unit process must be an inflow into the next unit process. Requires the existence of a unit process library file.

**Parameters**

- **chain_data** (*dataframe/str*) – Dataframe or filepath to a excel or tabular data that specifies the linkages of the unit process, with each row detailing an inflow, a unit process, and an outflow, with the outflow in one row being the inflow in the next row. Chain data is order depedent. Specifying the inflow of the first row or outflow of the last row is optional.
- **name** (*str*) – Name for the chain. Optional. (Defaults to "Product Chain.)
- **xls_sheet** (*str/None*) – Excel sheet where chain data resides. Optional. (Defaults to None)

**name**

the name of the chain

**process_chain_df**

the dataframe of chain linkages

**default_product**

if no other product is specified when balancing the chain, this will be used. Determined by process_chain_df: if there is an outflow specified for the last process or an inflow specified for the first process. If both are specified, the outflow is used by default.

**process_list**
list of unit process information generated using the process_chain_df. The list is in order of the unit processes in the chain, and each list item is a dictionary including the keys: - process: UnitProcess object - i: (str) inflow from process_chain_df - o: (str) inflow from process_chain_df

### 1.3.2 ProductChain.build()

`processchain.ProductChain.build(self)`
Generates the needed unit process objects

Chekcs that the processes specified in the chain dataframe have data available in the unit process library. Checks that the inflow and outflow specified in the chain exist for the associated unit process. Populates self.default_product and self.process_list.

### 1.3.3 ProductChain.balance()

`processchain.ProductChain.balance(self, qty, product=False, i_o=False, scenario=dat.default_scenario)`
Calculates the mass balance of the product chain

Based on a quantity of an inflow for the first unit process in the chain or an outflow for the last unit process, calculates the remaining quantities of all flows in the chain, assuming all unit processes are well-specified with zero degrees of freedom.

**Parameters**

- **qty** (*float*) – the quantity of the product to balance on.
- **product** (*str/bool*) – the product name. If False, uses the default product in the chain object attributes. Required if balancing a chain on an intermediate process. (Defaults to False)
- **i_o** (*str/bool*) – String beginning with "i" or "o". "i" if the product is an inflow of the chain's first unit process, "o" if it is an outflow of the chain's last unit process. If False, first checks to see if product exist in last unit process's outflows, then first unit process's inflows, and uses the first matching product that it finds. Required if balancing a chain on an intermediate process. (Defaults to False)
- **unit_process** (*str*) – Name of the unit process that the specified product belongs to. Necessary to balance chain on intermediate process.
- **scenario** (*str*) – The name of the scenario of variable values to use, corresponding to the matching row index in each unit process's var_df. (Defaults to the string specified in dat.default_scenario)

**Returns**

- 3-level nested dictionary of inflows, `[unit process][flowtype][substance] = (float)`
- 3-level nested dictionary of outflows, `[unit process][flowtype][substance] = (float)`
- dictionary of intermediate flows, `[substance] = (float)`
- list of lists of internal flows, `[chain, origin unit, product, qty (float), chain, destination unit]`

### 1.3.4 ProductChain.diagram()

`processchain.ProductChain.diagram(self, view_diagram=True, save=True, outdir=f'{dat.outdir}/pfd')`
Generates a diagram of the chain

Using Graphviz, takes the unit process names, sets of inflows and outflows, and the specified linkages of the chain to generate a diagram of the chain as a png and svg.

Code looks repetitive, but slightly different elements are needed to be generated if the unit process is the first in the list, a middle process, or the last unit process in the list, as well as a special case needed if the chain is only one unit process long. Each node is also given a long unique identifier, which is necessary for building factory-level diagrams.

The use of a product flow subgraph allows the unconnected inflows and outflows to appear in invisible (white) nodes, and also is returnable for use in larger factory diagrams.

**Parameters**

- **`view`** (*`bool`*) – If True, displays the diagram in the system viewer. (Defaults to True)
- **`save`** (*`bool`*) – If True, writes the diagram to file (Defaults to True)
- **`outdir`** (*`str/bool`*) – The output directory where to write the files. (Defaults to the output directory specified in dataconfig in a 'pfd' subfolder.)

**Returns** The Digraph object of the product flow diagram, with each unit process as a node, with the concatanated chain name and unit process name (e.g. chainunitprocess) as the identifier, and also the non-linking inflows and outflows as white-bordered nodes with concatanated chain, process, and flowtype (e.g. chainunitprocessinflows)

**Return type** Optional

## 1.4 The Factory Module

Note: This page has not yet been formatted. It is currently only the automodule documentation.

Factory class

This module contains the Factory class, which are objects that link together (and can generate) a set of product chains. One product chain produces the primary product(s) of the factory, whereas other auxillary chains provide inflows or processes outflows from the main chain or other auxillary chains. Auxillary chains can be attached to any process in any chain in the factory.

Module Outline:

- import statements and logger
- **class: Factory**
  - class function: Build
  - class function: Balance
  - class function: Diagram

**class** `factory.`**`Factory`**(*chain_list_file*, *connections_file=None*, *chain_list_sheet=None*, *connections_sheet=None*, *name='Factory'*)

Factories are sets of one or more connected products chains

Factories have a primary product chain, and are balanced on a specific outflow of that chain. Auxillary chains link to the main chain (or other auxillary chains) at any or multiple unit processes in the chain, and balance on a specified inflow or outflow of that unit process(es), i.e. it is assumed the auxillary chain is either providing the inflow or accepting the outflow.

Factories balance on a specified output product. Product chains that are not the main output product balance on the relevent inputs and outputs of the main product. All product chains in a factory are run using variables from the same scenario.

**Note:** The product chain (main chain) is assumed to be the first chain in the tabular data listing the chains in the factory.

**Parameters**

- **chains_file** (*DataFrame/str*) – Dataframe or filepath to tabular data detailing the process chains in the factory. The chain in the first row of data is assumed to be the main product flow. Must contain columns for: [Chain Name, Chain Product, Product_IO, ChainFile]
- **connections_file** (*DataFrame/str*) – Dataframe or filepath to tabular data detailing the connections between the chains in the factory. Must contain columns for: [OriginChain, OriginProcess, Product, Product_IO_of_Origin, Product_IO_of_Destination, DestinationChain]
- **name** (*str, optional*) – The name of the factory. Defaults to False.

**name**
The name of the factory.

**Type** str

**chains_df**
Tabular data of the factory chains and the location of their data.

**Type** data frame

**connections_df**
Tabuloar data of the connections between the factory chains.

**Type** data frame

**main_chain**
the name of the factory's product chain, taken from the first row of chains_df.

**Type** str

**main_product**
the name of the factory's main product, taken from the first row of chains_df.

**Type** str

**chain_dict**
Dictionary of dictionaries containing process chain objects in the factory. Each chain name is an entry key, with a value of a dictionary containing the process chain object, name, product, and whether that product is a chain inflow or outflow.

**Type** dict

**balance** (*main_product_qty, scenario='default', write_to_xls=True, outdir='BlackBlox_output', mass_energy=True, energy_flows=['heat', 'energy', 'electricity', 'power']*)
Calculates the mass balance of the factory

Based on a quantity of the factory's main product, calculates the remaining quantities of all flows in the factory, assuming all unit processes are well-specified with zero degrees of freedom.

**Parameters**

- **main_product_qty** (*float*) – the quantity of the product to balance on.
- **product** (*str/bool*) – the product name. If False, uses the default product in the chain object attributes. (Defaults to False)
- **scenario** (*str*) – The name of the scenario of variable values to use, corresponding to the matching row index in each unit process's var_df. (Defaults to the string specified in dat.default_scenario)
- **write_to_xls** (*bool*) – If True, outputs the balances to an excel workbook, with sheets for factory totals, inflows and outflows for all unit processes, and inflows and outflows by chain. (Defaults to True)

- **`outdir`** (*`str`*) – Filepath where to create the balance spreadsheets. (Defaults to the outdir specified in dataconfig)
- **`mass_energy`** (*`bool`*) – If true, seperates mass and energy flows within each excel sheet, adding rows for the respective totals. (Defaults to True)
- **`energy_flows`** (*`list`*) – list of prefix/suffixes used to identify which substances are energy flows and seperates them. (Defaults to dat.energy_flows)

**Returns** dictionary of factory total inflow quantities by substance dictionary of factory total outflow quantities by substance

**`build`**()

Generates the needed process chain objects for the factory

Locates the data specified for each process chain, and generates objects for each chain. Populates self.main_chain, self.main_product, and self.chain_dict.

**`diagram`**(*outdir='BlackBlox_output'*, *view=False*)

Outputs a diagram of the factory flows to file.

Using Graphviz, takes the unit process names, sets of inflows and outflows, and the specified linkages of the factory to generate a diagram of the chain as a png and svg.

**Parameters**

- **`outdir`** (*`str`*) – The output directory where to write the files. (Defaults to the output directory specified in dataconfig in a 'pfd' subfolder.)
- **`view`** (*`bool`*) – If True, displays the diagram in the system viewer. (Defaults to True)

## 1.5 The Industry Module

Note: This page has not yet been formatted. It is currently only the automodule documentation.

**class** `industry.`**`Industry`**(*factory_list_file*, *factory_list_sheet=None*, *name='Industry'*, ***kwargs*)

Industries are made up of one or more factories, and are balanced on one or more factory products. Factories within an industry can run with different scenario data. Industries can change over time.

**`balance`**(*products_data=None, products_sheet=None, mass_energy=True, energy_flows=['heat', 'energy', 'electricity', 'power'], force_scenario=None, write_to_xls=True, outdir='BlackBlox_output', subfolder=True, foldertime=True, file_id='', diagrams=True*)

Balances an industry using one scenario for each factory.

**`build`**()

generates the factory, chain, and process objects in the industry

**`evolve`**(*start_data=None, start_sheet=None, end_data=None, end_sheet=None, start_step=0, end_step=1, mass_energy=True, energy_flows=['heat', 'energy', 'electricity', 'power'], write_to_xls=True, outdir='BlackBlox_output', file_id='', diagrams=True, graph_outflows=False, graph_inflows=True*)

Calculates timestep and cumulative inflows and outflows of an industry using a specified starting scenario and end scenario

**`run_scenarios`**(*scenario_list*, *products_data=None*, *products_sheet=None*, *write_to_xls=True*, *outdir='BlackBlox_output'*, *file_id=''*, *diagrams=False*)

Balances the industry on a set of forced industry-wide scenarios

## 1.6 The Calculators Module

Note: This page has not yet been formatted. It is currently only the automodule documentation.

Calculator functions used in balancing unit processes

This module contains the standard set of functions used to manipulate the quantites specified in the user data. The calculation type specified by the user must be of one of the type specified here, or one defined by the user in custom_lookup.py

The "invert" option throughout the calculation helps the unit processes to balance processes balance on an arbitrary known quantity even though the user is specifying only a single-directional relationship between substance quantities in the relationships table.

Note, the use of **kwargs in the function argument calls is required to allow the functions to work properly, since all possible calculatr variables are provided to the calculator function in unitprocess.py, whether or not they are used by that specific function.

Module Outline: - imports and logger - function: check_qty - function: Ratio - function: Remainder - function: ReturnValue - function: MolMassRatio - function: check_balance - module variable: calcs_dict

`calculators.`**`Addition`**(*qty*, *qty2*, *invert=False*, ***kwargs*)

Adds one quantity to another.

Assuming the known values are X and X1, returns Y = X + X1. If invert is True, then assumes that the known values are Y and X1, and returns X = Y - X1

**Parameters**

- **`qty`** (*`float`*) – The quantity of a known substance
- **`qty2`** (*`float`*) – The quantity of another known substance
- **`invert`** – If True, subtracts rathers the values rather than adds them. (Defaults to False)

**Returns** The quantity of a third substance

**Return type** float

`calculators.`**`Combustion`**(*known_substance, qty, unknown_substance, var, emissions_dict=False, inflows_dict=False, emissions_list=['CO2', 'SO2', 'H2O'], fuels_df= HHV CO2 H2O fuel coal 33.70 3.19 0 charcoal 30.00 2.93 0 natural gas 40.00 2.75 0 diesel 45.30 3.15 0 wood chips 21.00 1.80 0 steam 2.77 0.00 1 coke 29.01 3.23 0 Eurofer electricity mix proxy 1.00 0.11 0, **kwargs*)

Calculates fuel or energy quantity and combustion emissions

This is a function designed to calculate the quantity of fuel necessary to generate a certain quantity of energy, or the quantity of energy generated by combusting a given quantity of fuel. It requires the existence of a dataframe with the HHV of the given fuel.

It can also calculate the emission of combustion, if relevent factors are present in the fuel dataframe, and output that data to a outflows dictionary (e.g that of the unit process calling the function).

Note: Besides the listed emissions, it will also calculate "waste heat" based on the combustion efficiency.

To maintain mass balances, the difference between the combustion emissions and the fuel mass is added to the inflows dictionary (if specified) as 'O2' (oxygen). To calculate non-oxygen-based combustion emissions and co-inflows, instead use the "stoichiometric_combustion" function, which requires the elemental composition of the fuel.

## Examples

```
>>> Combustion('charcoal', 3, 'heat', 0.8)
72.0
```

```
>>> Combustion('heat', 3, 'charcoal', 0.8)
0.125
```

**Parameters**

- **known_substance** (*str*) – The name of the known substance, either a solid fuel or an energy flow. If the string is not in the index of the fuels dataframe, known_substance is assumed to be the energy output of the combusted fuel.
- **qty** (*float*) – the quantity of the known substance (mass or energy)
- **unknown_substance** (*str*) – The name of the unknown substance, either a solid fuel or an energy flow. If the string is not in the index of the fuels dataframe, known_substance is assumed to be the energy output of the combusted fuel.
- **var** (*float*) – the efficiency of combustion (Between 0 and 1).
- **emissions_dict** (*defaultdict/bool*) – The destination dictionary where to store the calculated combustions emissions data. If no dictionary is provided, emission data will not be stored. (Defaults to False)
- **inflows_dict** (*defaultdict/bool*) – The destination dictionary where to store the calculated combustion co-inputs data. If no dictionary is provided, co-inputs data will not be stored. Currently the only co-input considered is oxygen. (Defaults to False)
- **emissions_list** (*list[str]*) – List of emissions to calculate, if emission factors are available in the fuel dataframe (Defaults to default_emission_list, defined above.)
- **fuels_df** (*dataframe*) – The dataframe containing the names, HHVs, and emissions data of the fuel. (Defaults to df_fuels)

**Returns** The quantity of the unkown substance (fuel or energy)

**Return type** float

calculators.**MolMassRatio**(*known_substance*, *qty*, *unknown_substance*, ***kwargs*)

Uses the ratio of molecular mass of two substances to

This function takes the quantity of a known substance and the name of that substance and another which are also valid chemical compositions and uses the ratio of their molecular weights to calculate the quantity of the substance of unknown quantity.

### Example

```
>>> MolMassRatio('CaCO3', 5, 'CO2')
2.198564447495127
```

```
>>> MolMassRatio('C8H10N4O2', 5, 'C12H22O11')
8.81341527344784
```

**Parameters**

- **known_substance** (*str*) – The name of the substance of known quantity, which must be a valid chemical composition
- **qty** – The quantity of the known substance

  **unknown_substance (str): The name of the substance of unknown quantity,**
  which must be a valid chemical composition.

calculators.**Ratio**(*qty*, *var*, *invert=False*, ***kwargs*)

Multiplies or divides a quantity by a given ratio.

This function exists as it is to allow unit processes to be calculated backwards or forwards. The user only has to specify one relationship (e.g. Y:X for a known X), but if it is Y that's known, then it flips the ratio to be X:Y.

### Examples

```
>>> Ratio(5, 3)
15
```

```
>>> Ratio(5, 3, invert=True)
1.6666666666666665
```

**Parameters**

- **qty** – The known quantity
- **var** – The ratio of unknown:known quantities
- **invert** (*Bool*) – Specigy True if the ratio is known:unknown. Converts the ratio to 1/ratio. (Default is False.)

**Returns** The original quantity multipled by the ratio.

calculators.**Remainder**(*qty*, *var*, *invert=False*, ***kwargs*)

Multiplies a quantity by the inverse of a known ratio between 0 and 1

The Remainder function is useful when both the known and unknown quantities are two parts of the same total, but the percentage balance may change (such as with efficiencies (product + loss will always equal 100%). The ratio of X:Y and Y:X should always equal 1.0.

### Examples

```
>>> Remainder(5, 0.3)
3.5
```

```
>>> Remainder(5, 0.3, invert=True)
7.142857142857143
```

**Parameters**

- **qty** – The known quantity
- **var** – The ratio of known:total quantities. A number between 0 and 1, where 1-var is the ratio of unknown:total.
- **invert** (*bool*) – True if the ratio is unknown:total. Converts the ratio to 1/ratio.

calculators.**ReturnValue**(*qty*, ***kwargs*)

Returns the value passed to it.

Returns the value passed as the qty argument. Useful for creating temporary values with unique names in the unit process's temporary dictionary if substance of same name exists in both input and output dictionary.

### Examples

```
>>> ReturnValue(5)
5
```

**Parameters** **qty** – a quantity (or anything really)

**Returns** literally the exact thing you gave it.

**Return type** qty

`calculators.`**`Subtraction`**(*qty*, *qty2*, *invert=False*, ***kwargs*)

Subtracts one quantity from another.

Assuming the known values are X and X1, returns Y = X - X1. If invert is True, then assumes that the known values are Y and X1, and returns X = Y + X1

**Parameters**

- **`qty`** (*`float`*) – The quantity of a known substance
- **`qty2`** (*`float`*) – The quantity of another known substance
- **`invert`** – If True, adds rathers the values rather than subtracts them. (Defaults to False)

**Returns** The quantity of a third substance

**Return type** float

`calculators.`**`calcs_dict = {'addition': <function Addition>, 'combustion': <function Com`**

Dictionary of calculators available to process unit process relationships. Must be manually updated if additional calculators are added to this module. Automatically pulls in calculators in the custom_lookup module, if they are specified in the custom_calcs_dict in that module.

Used by the Unit Process class's balance function.

`calculators.`**`check_balance`**(*inflow_dict*, *outflow_dict*, *raise_imbalance=True*, *ignore_flows=[]*, *only_these_flows=False*, *round_n=5*)

Checks whether two dictionary mass values sum to equivelent quantities

Used to check whether the inflows and outflows of a unit process are balanced. Takes dictionaries of inflows and outflows with substance names as keys and quantities as values. Optionally can use a list of substance name prefix/suffix tags to exclude flows (e.g. energy flows in a mass balance), or only include certain flows (e.g. only include those with the energy tag)

**Parameters**

- **`inflow_dict`** (*`dict/defaultdict`*) – the dictionary of inflows with the format inflow_dict[substance name] = quantity
- **`outflow_dict`** (*`dict/defaultdict`*) – the dictionary of outflows with the format outflow_dict[substance name] = quantity
- **`round_n`** (*`int`*) – Number of places after the decimal to use when checking if inflow and outflow masses are equivelent. Defaults to 5.
- **`ignore_flows`** (*`list[str]`*) – A list of strings that indicate that a substance is not be included in the balance. If any of the keys in he inflow_dict or outflow_dict begin or end with a string in the ignore_flows list, the substance will not be added to the total quantity. (Defaults to an empty list)
- **`only_these_flows`** (*`bool/list`*) – if given a list of strings, will only include those flows that begin or end with the strings in this

**Returns** True if the mass flows are balanced. Otherwise False. float: sum of all mass inflows. float: sum of all mass outflows.

**Return type** bool

`calculators.`**`check_qty`**(*qty*, *fraction=False*)

Checks that a quantity is > 0 and optionally < 1. Raises and error if the quantity is negative, and, optionally, if it is less than one. Currently BlackBlox does not support negative masses or energy contents. This isn't quantum.

**Parameters**

- **`qty`** (*`float or int`*) – number to check
- **`fraction`** (*`bool`*) – whether the number should be between 0 and 1 (Defaults to False.)

`calculators.`**`twoQty_calc_list = ['subtraction', 'addition']`**

List of calculations that require two quantities to exist in the unit process flow dictionary.

Used by the Unit Process class's balance function.

## 1.7 The IO Functions Module

Note: This page has not yet been formatted. It is currently only the automodule documentation.

Data input and output manipulation functions

This module contains data manipulation functions used in BlackBlox.py to retrieve user input and manipulate it into useable formats, as well as functions used in output file generation.

Module Outline: - imports - function: make_df - function: check_for_col - function: build_filedir - function: write_to_excel - fucntion: sl - function: if_str - function: nested_dicts

`io_functions.`**`build_filedir`**(*filedir*, *subfolder=None*, *file_id_list=[]*, *time=True*)

Builds complicated file directory names. Used for allowing file to be output to unique directories.

**Parameters**

- **`filedir`** (*`str`*) – the base file directory
- **`subfolder`** (*`str`*) – optional subfolder(s) (Defaults to None.)
- **`file_id_list`** (*`list of str`*) – list of strings to append to the directory name. (Defaults to an empty list.)
- **`time`** (*`bool`*) – Whether to include a date-time stamp in the file directory name. (Defaults to True.)

**Returns** The built file directory string.

`io_functions.`**`check_for_col`**(*df*, *col*, *index*)

Checks if a column exists and returns column value at the given index

This function is used mostly to check for whether a column for excel worksheet data exists in a particular data location list data frame.

**Parameters**

- **`df`** (*`pandas.dataframe`*) – The data frame to check
- **`col`** (*`str`*) – the column name to check for
- **`index`** (*`str or int`*) – the row index to get column value for, if column exists.

**Returns** Value at index, column in the dataframe, if column exists, otherwise returns None

`io_functions.`**`clean_str`**(*string_to_check*, *str_to_cut=False*, *lower=True*, *remove_dblnewline=True*)

Multipurpose function to clean user input strings

Used to clean user input. First creates a copy of the string to prevent the original string from being modified unintentionally.

**Parameters**

- **`string_to_check`** (*`str`*) – The string to strip and lower
- **`str_to_cut`** (*`str/list/bool`*) – If passed a string, will check for and remove the cut_str string from the string_to_check. If passed a list of strings, will check for and remove each from the string_to_check. (Defaults to False.)
- **`remove_dblnewlines`** (*`bool`*) – If True, looks for and removes instances of double new lines. (Defaults to True.)

**Returns** The input string, stripped of leading and trailing white space and in all lowercase.

`io_functions.`**`if_str`**(*maybe_a_string*)
checks if an input is a string, returns string if so, None if not.

**Parameters** **`string`** – a thing to check if it's a string.

**Returns** The string, if it was a string, otherwise returns None.

`io_functions.`**`is_energy`**(*string*, *energy_strings=dat.energy_flows*)
checks if a string refers to an energy flow

**Parameters**

- **`string`** (`str`) – the string to check
- **`energy_strings`** (`list`) – list of strings that mark whether the substance name that starts or ends with them is an energy flow.

**Returns** bool. True, if the string starts or ends with the energy marker. Otherwise, False.

`io_functions.`**`make_df`**(*data*, *sheet=None*, *sep='\t'*, *index=0*, *metaprefix='meta'*, *col_order=False*, *T=False*, *drop_zero=False*, *sort=False*)
Creates a Pandas dataframe from various file types

Numbers that are initially read as strings will be converted to numeric values when possible (errors are ignored).

**Parameters**

- **`data`** – accepts both objects that can be made into dataframes (e.g. nested) dictionaries and strings of filespaths (including excel workbooks, comma seperated value files, and other delimited text files)
- **`sheet`** (`str, optional`) – The worksheet of a specified excel workbook. (Defaults to None)
- **`sep`** (`str`) – the seperator used in non-csv text file. (Defaults to tab ( ))
- **`index`** (`int or None`) – the column of data that is the index. (Defaults to 0)
- **`metaprefix`** (`str/None`) – If a column name or row index begins with the metaprefix, that row or column is dropped from the data frame. (Defaults to 'meta')
- **`col_order`** (`list[str]/False`) – If a list is passed, will use those strings as the column names of the dataframe in the order in the list.
- **`T`** (`bool`) – If True, transposes the data frame before return. (Defaults to False)
- **`drop_zero`** (`bool`) – If True, converts any NaNs to zeros, and then removes any rows or columns that contain only zeros.
- **`sort`** (`bool`) – Whether to sort the data by the ascending order of the index

**Returns** The generated dataframe.

`io_functions.`**`mass_energy_df`**(*df*, *energy_strings=['heat', 'energy', 'electricity', 'power']*, *totals=True*)
Reorders dataframe to seperate mass and energy flows

Uses a list of prefix/suffixes to identify mass and energy flows and reorders a dataframe to seperate them, orders them alphabetically, and also optionally adds rows for mass totals and energy totals.

**Parameters**

- **`energy_strings`** (`list`) – contains strings of prefix/suffix to substance names that indicate an energy flow (Defaults to dat.energy_flows)
- **`totals`** (`bool`) – Appends summation rows for mass and energy seperately. (Defaults to True)

`io_functions.`**`nested_dicts`**(*levels=2*, *final=<class 'float'>*)
Created a nested defaultdict with an arbitrary level depth

### Example

1_level_dict[1st] = final (Same as standard defaultdict) 2_level_dict[1st][2nd] = final 4_level_dict[1st][2nd][3rd][4th] = final

source: https://goo.gl/Wq5kLq

**Parameters**

- **levels** (*int*) – The number of nested levels the dictionary should have. Defaults to 2. (As 1 is just a normal defaultdict)
- **final** (*type*) – The type of data stored as the value in the ultimate level of the dictionary. Defaults to float because that's what most commonly used in this package.

**Returns** A nested defaultdict of the specified depth.

`io_functions.`**`write_to_excel`**(*df_or_df_list*, *sheet_list=None*, *filedir='BlackBlox_output'*, *filename='output'*)

Writes one or more data frames to a single excel workbook.

Each data frame appears on its own worksheet. Automatically creates the specified output folder if it does not exist.

**Parameters**

- **df_or_df_list** (*dataframe or list*) – A single pandas dataframe or list of pandas dataframes.
- **sheet_list** (*optional, list*) – List of sheetnames that will be used for the dataframe at the same index in df_or_df_list. (Defaults to None.)
- **filedir** (*optional, str*) – desired file output directory. (Defaults to current working directory.)
- **filename** (*optional, str*) – desired excel file name, without extension. (Defaults to 'output')

## 1.8 The dataconfig Module

Note: This page has not yet been formatted. It is currently only the automodule documentation.

Data Input and Output default strings

This module contains variables used for default file paths, column headers for input data files, as well as special variable strings (generally used) to indicate whether something is All or None.

Please note that as of this time, these variables are case and white space sensitive.

Module Outline:

- import statements and logger
- module variables: default filepaths (str)
- module variables: user data (dict)
- **module variables: column headers (str)**
  - for unit process library tabular data
  - for unit process relationships tabular data
  - for product chain linkages tabular data
  - for factory product chain list tabular data
  - for factory linkages tabular data
  - for industry data

- **module variables: reserved words for tabular data**
  - same_xls (list)
  - default_scenario (str)
  - no_var (list)
  - massless_flows (list)
  - connect_all (str)
  - all_factories (list)

`dataconfig.`**`all_factories = ['industry', 'all', 'factories']`**

strings indicating all factories in a given industry

Useable as a row index in industry scenario tables to indicate that a production quantity or scenario applies to all factories producing the specified product.

If used to specify an industry-wide total product production quantity, each factory producing that product should specify their production quantity as a fraction of that total as a decimal between 0 and 1.

**Type** list

`dataconfig.`**`connect_all = 'all'`**

indicator that all processes of a chain connect to the destination

Usable in the factory connections table, for the “origin process” column. Indicates that every process of the origin chain is connected to the destination chain by the specified product.

**Type** str

`dataconfig.`**`default_scenario = 'default'`**

the index used for the default scenario of variables

Usable in the unit process variables data tables. If present in the variables data index, the default scenario will be used when a scenario of variables is not otherwise specified.

**Type** str

`dataconfig.`**`energy_flows = ['heat', 'energy', 'electricity', 'power']`**

indicator that a substance is an energy flow

Usable in flow names. Must be used at the beginning or end of the flow name.

**Type** list

`dataconfig.`**`lookup_var_dict = {'fuel':  {'filepath':  '/Users/Tanzer/GitHub/BlackBlox/exc`**

dictionary of special lookup substance names Lookup_var_dict is a dictionary with the names of substance, that when used in the unit process calculations file, will trigger the program to replace the lookup substance name with the substance name specified in the unit process’s variable data table for the scenario currently in use.

Each entry in this dictionary should be formatted with the following:

**key** *(str)*: the substance name to be used in the calcuations file

**value** ***(dict)*****: a dictionary of lookup variable attributes, containing: lookup_var** *(str)*: the header of the column in the unit process variable file that contains the value with which to replace the lookup substance word.

**data_frame** *(optional)*: a data frame with additional custom data about the lookup variable, such as to be used in custom functions, below. These are not used elsewhere in BlackBlox.py.

`dataconfig.`**`no_var = ['none', 'false', 'na', '-', '--', '', 'nan', 0, '0', None, False, r`**

indicator that no variable is used in the calculation

Usable in the unit process calculation table, to indicate that the calculation type requires no variable beyond the names of the substances. (e.g. MolMassRatio)

**Type** str

`dataconfig.`**`outdir = 'BlackBlox_output'`**

The default output directory.

Unless an absolute path is specified, BlackBlox will create the directory as a subfolder of the current working directory.

**Type** str

`dataconfig.`**`same_xls = ['thisfile', 'same', 'here']`**

strings indicating the data is in the current Excel workbook

Usable as a replacement for a filepath for input data that is in an Excel workbook with multiple sheets. The correct Excel sheet must still be specified.

**Type** list

`dataconfig.`**`set_config`**`(`*config_data*, *user_data*`)`

Allows user to specify config data without changing module default values

`dataconfig.`**`unit_process_library_file = '/Users/Tanzer/GitHub/BlackBlox/excelData/globalD`**

The filepath whre the unit process library file exists.

**Type** str

`dataconfig.`**`unit_process_library_sheet = 'Unit Processes'`**

The worksheet of the unit process library, if in an Excel workbook

If not an excel worksheet, this variable should be None.

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX