bincfg.utils package

This subpackage contains various utility functions.

Submodules

bincfg.utils.cfg_utils module

Utilities for CFG/MemCFG objects and their datasets

`bincfg.utils.cfg_utils.`**`check_for_normalizer`**(*dataset*, *cfg_data*)
Checks the incoming data for a normalizer to set to be *dataset*'s normalizer

Assumes this dataset does not yet have a normalizer. Searches the incoming *cfg_data* for a cfg/dataset that has a normalizer, and sets it to be this dataset's normalizer. If this method finds no normalizer, or multiple unique normalizers, then an error will be raised.

Parameters:
- **dataset** (*Union[CFGDataset, MemCFGDataset]*) –
- **cfg_data** (*Iterable[Union[CFG, MemCFG, CFGDataset, MemCFGDataset]]*) –
- **dataset** – a `CFGDataset` or `MemCFGDataset` without a normalizer
- **cfg_data** – an iterable of `CFG`/`MemCFG`/`CFGDataset`/`MemCFGDataset`'s

Raises: **ValueError** – when there are multiple conflicting normalizers, or if no normalizer could be found

`bincfg.utils.cfg_utils.`**`get_address`**(*obj*)
Gets the integer address from the given object

Parameters: **obj** (*Union[str, int, Addressable]*) – a string, int, or object with a string/int *.address* attribute (should always be positive)

Raises:
- **TypeError** – *obj* is an unknown type
- **ValueError** – given address is negative

Returns: the integer address

Return type: int

`bincfg.utils.cfg_utils.`**`get_special_function_names`**()
Returns the current global special function names

`bincfg.utils.cfg_utils.`**`update_atomic_tokens`**(*file_tokens*, *curr_data*, *update_tokens*)
Updates atomic tokens. Only meant to be passed to AtomicData.atomic_update as the function to use

`bincfg.utils.cfg_utils.`**`update_memcfg_tokens`**(*cfg_data*, *tokens*)
Adds all new tokens to *tokens*, and updates all tokens in *cfg_data* to their respective values in *tokens*

Tokens in *cfg_data* will be modified, as will the *.asm_lines* attribute of each memcfg. Assumes the *cfg_data* has conflicting tokens to *tokens* and thus needs modification. Both *cfg_data* and *tokens* will be modified in-place.

Parameters:
- **cfg_data** (*Union[MemCFG, MemCFGDataset]*) – the memcfg/memcfgdataset to have its tokens changed
- **tokens** (*Union[Dict[str, int], AtomicData]*) – the dictionary of tokens to update with the new tokens in *cfg_data*. Can be an AtomicData object for atomic updating of tokens

bincfg.utils.misc_utils module

Miscellaneous utility functions

exception `bincfg.utils.misc_utils.`**`EqualityCheckingError`**
Bases: `Exception`

Error raised whenever there is an unexpected problem attempting to check equality between two objects

exception `bincfg.utils.misc_utils.`**`EqualityError`**(*a*, *b*, *message=None*)
Bases: `Exception`

Error raised whenever an `equal()` check returns false and *raise_err=True*

`bincfg.utils.misc_utils.`**`arg_array_split`**(*length*, *sections*, *return_index=None*, *dtype=<class 'numpy.uint32'>*)
Like np.array_split(), but returns the indices that one would split at

This will always return *sections* sections, even if *sections* > length (in which case, any empty sections will come at the end). If *sections* does not perfectly divide *length*, then any extras will be front-loaded, one per split array as needed.

NOTE: this code was modified from the numpy array_split() source

Parameters:
- **length** (*int*) – the length of the sequence to split
- **sections** (*int*) – the number of sections to split into
- **return_index** (*Optional[int]*) – if not None, then an int to determine which tuple of (start, end) indices to return (IE: if you were splitting an array into 10 sections, and passed return_index=3, this would return the tuple of (start, end) indicies for the 4th split array (since we start indexing at 0))
- **dtype** (*np.dtype*) – the numpy dtype to use for the returned array

Returns: a numpy array of length *sections + 1* where the split array at index *i* would use the start/end endices *[returned_array[i]:returned_array[i+1]]*, unless return_index is not None, in which case a 2-tuple of the (start_idx, end_idx) will be returned

Return type: Union[np.ndarray, Tuple[int, int]]

`bincfg.utils.misc_utils.`**`eq_obj`**(*a*, *b*, *selector=None*, *strict_types=<object object>*, *unordered=<object object>*, *raise_err=<object object>*)

Determines whether a == b, generalizing for more objects and capabilities than default __eq__() method. Equal() is an equivalence relation, and thus:

1. equal(a, a) is always True (reflexivity)
2. equal(a, b) implies equal(b, a) (symmetry)
3. equal(a, b) and equal(b, c) implies equal(a, c) (transitivity)

NOTE: This method is not meant to be very fast. I will apply as many optimizations as feasibly possible that I can think of, but there will be various inefficient conversions of types to check equality.

NOTE: kwargs passed to the initial `equal()` function call will be passed to all subcalls, including those done in other objects using their built-in __eq__ function. Any objects can override those kwargs for any later subcalls (but not those above/adjacent). NOTE: The *selector* kwarg is only used once, then consumed for any later subcalls

Parameters:
- **a** (*Any*) – object to check equality
- **b** (*Any*) – object to check equality
- **selector** (*Optional[str]*) – if not None, then a string that determines the 'selector' to use on both objects for determining equality. It should start with either a letter (case-sensitive), underscore '_', dot '.' or bracket '['. This string will essentially be appended to each object to get some attribute to determine equality of instead of the objects themselves. For example, if you have two lists, but only want to check if their element at index '2' are equal, you could pass *selector='[2]'*. This is useful for debugging purposes as the error messages on unequal objects will be far more informative. Defaults to None. NOTE: if you pass a *selector* string that starts with an alphabetical character, it will be assumed to be an attribute, and this will check equality on *a.SELECTOR* and *b.SELECTOR*
- **strict_types** (*bool*) – if True, then the types of both objects must exactly match. Otherwise objects which are equal but of different types will be considered equal. Defaults to False.
- **unordered** (*bool*) – if True, then all known sequential objects (list, tuple, numpy array, etc.) will be considered equal even if elements are in a different order (eg: a multiset equality). Otherwise, sequential objects are expected to have their subelements appear in the same order. If the passed objects are not sequential, then this has no effect. Defaults to False.
- **raise_err** (*bool*) – if True, then an `EqualityError` will be raised whenever *a* and *b* are unequal, along with an informative stack trace as to why they were determined to be unequal. Defaults to False.

Raises:
- **EqualityError** – if the two objects are not equal, and *raise_err=True*
- **EqualityCheckingError** – if there was an error raised during equality checking

Returns: True if the two objects are equal, False otherwise

Return type: bool

`bincfg.utils.misc_utils.`**`eq_obj_err`**(*obj1*, *obj2*)

Same as eq_obj, but always raises an error

`bincfg.utils.misc_utils.`**`get_module`**(*package*, *raise_err=True*, *err_message=''*)

Checks that the given package is installed, returning it, and raising an error if not

Parameters:
- **package** (*str*) – string name of the package
- **raise_err** (*bool, optional*) – by default, this will raise an error if attempting to load the module and it doesn't exist. If False, then None will be returned instead if it doesn't exist. Defaults to True.
- **err_message** (*str*) – an error message to add on to any import errors raised

Raises: **ImportError** – if the package cannot be found, and *raise_err=True*

Returns: the package

Return type: Union[ModuleType, None]

bincfg.utils.misc_utils.**get_smallest_np_dtype**(*val*, *signed=False*)
Returns the smallest numpy integer dtype needed to store the given max value.

Parameters:
- **val** (*int*) – the largest magnitude (furthest from 0) integer value that we need to be able to store
- **signed** (*bool, optional*) – if True, then use signed ints. Defaults to False.

Raises: **ValueError** – if a bad value was passed, or if the value was too large to store in a known integer size

Returns: the smallest integer dtype needed to store the given max value

Return type: np.dtype

bincfg.utils.misc_utils.**hash_obj**(*obj*, *return_int=False*)
Hashes the given object

Parameters:
- **obj** (*Any*) – the object to hash
- **return_int** (*bool, optional*) – by default this method returns a hex string, but setting return_int=True will return an integer instead. Defaults to False.

Returns: hash of the given object

Return type: Union[str, int]

bincfg.utils.misc_utils.**isinstance_with_iterables**(*obj*, *types*, *recursive=False*, *ret_list=False*)
Checks that obj is one of the given types, allowing for iterables of these types

Parameters:
- **obj** (*Any*) – the obj to test type
- **types** (*Union[type, Tuple[type, ...]]*) – either a type, or tuple of types that obj can be
- **recursive** (*bool, optional*) – by default, this method will only allow iterables to contain objects of a type in *types*. If *recursive=True*, then this will accept arbitrary-depth iterables of types in *types*. Defaults to False.
- **ret_list** (*bool, optional*) – if True, will return a single list of all elements (or None if the isinstance check fails). Defaults to False.

Returns: the return value

Return type: Union[List[Any], bool, None]

bincfg.utils.misc_utils.**log**(*severity='info'*, *message=''*)
Attempts to log a message if LOGGER has already been set

bincfg.utils.misc_utils.**progressbar**(*iterable*, *progress=True*)
Allows one to call progressbar(iterable, progress) to determine use of progressbar automatically.

Checks to see if we are in a python notebook or not to determine which progressbar we should use. Copied from: https://stackoverflow.com/questions/15411967/how-can-i-check-if-code-is-executed-in-the-ipython-notebook

bincfg.utils.misc_utils.**scatter_nd_numpy**(*target*, *indices*, *values*)
Sets the values at *indices* to *values* in numpy array *target*

Shamelessly stolen from: https://stackoverflow.com/questions/46065873/how-to-do-scatter-and-gather-operations-in-numpy

Parameters:
- **target** (*np.ndarray*) – the target ndarray to modify
- **indices** (*np.ndarray*) – n-d array (same ndim as target) of the indices to set values to
- **values** (*np.ndarray*) – 1-d array of the values to set

Returns: the resultant array, modified inplace

Return type: np.ndarray

bincfg.utils.misc_utils.**set_logger**(*logger*)
Sets the logger for this module

bincfg.utils.misc_utils.**timeout_wrapper**(*timeout=3*, *timeout_ret_val=None*)

Wraps a function to allow for timing-out after the specified time. If the function has not completed after timeout seconds, then the function will be terminated.

bincfg.utils.mp_utils module

Utility functions involving multiprocessing

Contains functions for creating, getting, and terminating a main thread pool, as well as helpers to more easily map data to multiprocessing function calls.

exception bincfg.utils.mp_utils.**AquireLockError**(*attempts*, *lock_path*)
Bases: Exception

class bincfg.utils.mp_utils.**AtomicData**(*init_data*, *filepath=None*, *lock_path=None*, *max_read_attempts=None*)
Bases: object

A class that allows for atomic reading/updating of the given data to a pickle file

Parameters:
- **init_data** (*Any*) – Data to initialize the atomic file with. If the atomic file already exists, then that data will be loaded
- **filepath** (*Optional[str]*) – An optional filepath to store the dictionary, otherwise will be stored at ‘./atomic_dict.pkl’
- **lockpath** (*Optional[str]*) –

  An optional filepath for the lock file to use to atomically update the dictionary, otherwise will be stored at ‘./.[filepath].lock’ where [filepath] is the given *filepath* parameter

- **max_read_attempts** (*Optional[int]*) –

  An optional integer specifying the maximum number of attempts to atomically read this dictionary before giving up and raising an error. Set to None to attempt indefinitely. Defaults to None

aquire_lock()
Aquires the lock needed to update data

NOTE: this will prevent any and all updates to the atomic file until self.release_lock() is called. Make sure you call it quickly or other processes may hang!

NOTE: if the lock has already been aquired, nothing will happen

NOTE: it can be dangerous to attempt to aquire locks yourself, as any errors raised must be handled nicely and self.release_lock() must be called otherwise other processes may hang

atomic_read(*default=<object object>*)
Atomically reads the data from file, updating self.data

Parameters: **default** (*Optional[Any]*) – If this is passed and the file does not already exist, then this data will be saved to file and set to self.data

atomic_update(*update_func*, **update_args*, ***update_kwargs*)
Atomically updates the data

Will first aquire a lock on the data, read it in, then call *update_func(file_data, update_data)* where *file_data* is the data from the current atomic file, then write the data back to file and finally release the lock.

NOTE: this will prevent any and all updates to the atomic file until update_func has completed

NOTE: any errors within the update_func will be handled properly and will likely not mess up the atomic file

Parameters:
- **update_func** (*Callable*) – function that takes in: the data currently saved in file, the current data, then the passed args and kwargs, and returns the updated data to write back to file
- **update_args** (*Any*) – args to pass to update_func, after the current data saved in file
- **update_kwargs** (*Any*) – kwargs to pass to update_func

Returns: the updated data

Return type: Any

atomic_write()
Atomically writes the data at self.data to the pickle file

delete_file()
Atomically deletes the file being used

release_lock()
Releases the lock. Assumes it has already been aquired, otherwise an error will be raised

class bincfg.utils.mp_utils.**AtomicTokenDict**(*init_data=None*, ***atomic_data_kwargs*)
Bases: object

Acts like a normal token dictionary, but allows for atomic operations

addtokens(**tokens*)
Adds the given tokens to this dictionary, ignoring any that already exist

Parameters: **tokens** (*str*) – arbitrary number of string tokens to add to this token dict

property **data**
Returns the token dictionary

delete_file()
Deletes the atomic token dictinoary file

get(*key*, *default=None*)

items()

keys()

setdefault(*key*, *default=None*)
If the key exists, return the value. Otherwise set the key to the given default (or len(self) if default=None)

update(*tokens*)
Updates this dictionary with the given tokens

Parameters: **tokens** (*Dict[str, int]*) – dictionary mapping token strings to their integer values. Any tokens in the dictionary that are not in this dictionary will be added, and any tokens that already exist and have the same value will be ignored. If there are any tokens that already exist, but have a different value, then an error will be raised

values()

exception bincfg.utils.mp_utils.**MultiprocessingError**(*err*, *tb*)
Bases: Exception

Custom Error raised after an error in a multiprocessing call

class bincfg.utils.mp_utils.**RedirectStdStreams**(*stdout=None*, *stderr=None*)
Bases: object

Context manager to temporarily redirect stdout and stderr streams

NOTE: the passed streams will NOT be closed on exit

class bincfg.utils.mp_utils.**ThreadPoolManager**(*terminate=False*, *num_workers=None*)
Bases: object

A custom context manager to handle the global thread pool.

By default, this will only terminate the thread pool when an error was raised, however you can pass *terminate=True* to always terminate the thread pool after exiting this context. You can get the thread pool and number of workers with something like:

```
with ThreadPoolManager(num_workers=10) as tpm:
    pool = tpm.pool
    num_workers = tpm.num_workers
```

Parameters:

- **terminate** (*bool*) – if True, will terminate the thread pool after exiting, even if no error was raised
- **num_workers** (*Optional[int]*) –

  the number of workers to pass to get_thread_pool()

  NOTE: will only determine the number of workers if the thread pool as not yet been initialized. Otherwise the number of workers will be whatever it was previously set to

num_workers = *None*
the number of processes in the pool

pool = *None*
the multiprocessing process pool

bincfg.utils.mp_utils.**get_thread_pool**(*ret_n=False*, *num_workers=None*)
Returns (starts if needed) the current thread pool.

Parameters:
- **ret_n** (*bool, optional*) – if True, also returns the number of workers. Defaults to False.
- **num_workers** (*Union[int, None], optional*) – if not None, then the number of workers to use when initializing the thread pool. Only used if the thread pool is not currently initialized. Defaults to None.

Returns: the current global multiprocessing.Pool() object, or tuple of (pool, num_workers) if *ret_n=True*

Return type: Union[Pool, Tuple[Pool, int]]

`bincfg.utils.mp_utils.`**`init_thread_pool`**(*num_workers=None*)

Initializes the thread pool.

Parameters: **num_workers** (*int, optional*) – number of processes. Will default to min(os.cpu_count(), MAX_DEFAULT_NUM_WORKERS) if None. Defaults to None.

`bincfg.utils.mp_utils.`**`map_mp`**(*func*, *items=None*, *chunks=None*, *starmap=False*, *chunksize=1*, *num_workers=None*, *terminate=False*, *on_error='raise'*, *progress=False*)

Splits the given inputs over multiple processes in the global process pool

Maps the given items or chunks of items to the given function using multiprocessing, allowing for the use of a progressbar. Also handles terminating the pool.

Parameters:
- **func** (*Callable*) – the function to call
- **items** (*Iterable[Any], optional*) – an iterable of items to map. Mutually exclusive with chunks. Defaults to None.
- **chunks** (*Iterable[Iterable], optional*) – an iterable of iterables to map. Will send the entire chunk to another process and call the given function on each element, then return the results unpacked into a single total list. Mutually exclusive with items. Defaults to None.
- **starmap** (*bool, optional*) – if True, will call *starmap()* instead of *map()* on the pool. This assumes each element should be star-unpacked when sending to *func()*. Defaults to False.
- **chunksize** (*int, optional*) – if > 1, then multiple elements will be passed in one 'chunk' to each process. This would have the same effect (if using items) as passing chunks with this chunksize, but if using chunks, then this many chunks will be passed to each process instead. Defaults to 1.
- **num_workers** (*int, optional*) –

  the num_workers to pass to `bincfg.utils.get_thread_pool()`. Defaults to None.

  NOTE: this only has an effect if the thread pool has not yet been initialized. Otherwise this will use the num_workers the thread pool has already been initialized with.
- **terminate** (*bool, optional*) – if True, then the global thread pool will be terminated after use. If False, then the thread pool will only be terminated if an error is raised. Defaults to False.
- **on_error** (*_MpOnErrorType, optional*) –

  what to do when there is an error can be either a string describing what to do, or a callable in which case the callable will be called and its return value will be used in place of the error. Callables should take as input the error_object and traceback in that order as args, and return whatever value should be used in place of the errored value. Acceptable strings:
  - 'raise': raise any errors immediately
  - 'error_info': return the error information in place of the expected output whenever there is an error. Error information will be a 2-tuple of (error_object, traceback)
  - 'return_none': will return None in place of values whenever an error occurs, and the error will be ignored
  - 'ignore': ignore any errors and doesn't return any values for those which fail.

  Defaults to 'raise'.
- **progress** (*bool, optional*) – if True, will show a progressbar. Defaults to False.

Raises:
- **TypeError** – If none or both of *items* and *chunks* are passed
- **MultiprocessingError** – If there was an error running the multiprocessing calls, and *on_error='raise'*

Returns: a list of results

Return type: List[Any]

`bincfg.utils.mp_utils.`**`terminate_thread_pool`**(*send_sigint=False*)

Terminates the thread pool by first sending SIGINT, waiting for a few seconds, then SIGTERM if any are still running

Code from: https://stackoverflow.com/questions/47553120/kill-a-multiprocessing-pool-with-sigkill-instead-of-sigterm-i-think

Module contents