# Namespace `Napshift`

## Sub-modules

- Napshift.napshift
- Napshift.src

# Module `Napshift.napshift`

## Functions

### Function `main`

```
def main(
    pdb_file=None,
    pH=None,
    output_path=None,
    random_coil_path=None,
    talos_fmt=True,
    all_models=False,
    verbose=False
)
```

Main function that wraps the call of the predict method. First we create a ChemShiftPredictor object. We assume that the trained models are located in a folder named "/models/" that exists inside the parent directory (i.e. the same directory where the current script is stored). The output of the prediction is stored in the "output_path/" directory using a predetermined name that includes the PDB-ID from the input file.

:param pdb_file: This is the input (PDB) file that we want to predict the chemical shift values.

:param pH: The pH value (default is set to 7).

:param output_path: The directory (path) where we want to store the output file.

:param random_coil_path: If we have available random coil values for the same input file, we can use it here. If not (the default) we will generate automatically new values using the camcoil engine module.

:param talos_fmt: This is the TALOS format for the output file. If not set to True, than we will use a tabular format, where each row corresponds to a single residue and each atom has each own column.

:param all_models: If "True" the method will process all the models in the PDB file, otherwise only the first model.

:param verbose: This is a boolean flag that determines whether or not we want to display additional information on the screen.

:return: None.

# Namespace `Napshift.src`

## Sub-modules

- Napshift.src.chemical_shifts
- Napshift.src.random_coil
- Napshift.src.tests

# Module `Napshift.src.chemical_shifts`

## Sub-modules

- Napshift.src.chemical_shifts.auxiliaries
- Napshift.src.chemical_shifts.hydrogen_bond
- Napshift.src.chemical_shifts.input_vector
- Napshift.src.chemical_shifts.model_machine
- Napshift.src.chemical_shifts.order_parameters
- Napshift.src.chemical_shifts.ring_effect

# Module `Napshift.src.chemical_shifts.auxiliaries`

Constants that are used by all code are placed here for easy access. These include:

```
1. ACCEPTED_RES
2. RES_3_TO_1
3. TARGET_ATOMS
4. RANDOM_COIL_TBL
5. RANDOM_COIL_AVG
6. RANDOM_COIL_STD
7. ONEHOT (code: 10)
8. BLOSUM (code: 45, 50, 62, 80, 90)
```

## Functions

### Function `get_sequence`

```
def get_sequence(
    model,
    modified=False
)
```

Returns the amino-acid sequence (one letter code) for the given input model. Optionally, we can request the "modified" sequence version where the CYS and PRO res are distinguished in:

```
1) CYR (C) / CYO (X)

2) PRT (P) / PRC (O)
```

WARNING: The modified version assumes that the input model has already been processed with modify_models_OX. This will ensure that the b-factors are updated to (-1 / +1). Otherwise all CYS will be renamed to CYR and all PRO to PRT (which are the default values).

:param model: amino-acid chain (model) from biopython.

:param modified: (bool) Flag to indicate if we want to return the standard (20 : amino-acid), or the modified with the CYS: CYR / CYO and PRO: PRT / PRC distinction.

:return: A dictionary with {"A": "seq_A", "B": "seq_B", "C": "seq_C",...} pairs in one letter code.

### Function `modify_models_OX`

```
def modify_models_OX(
    structure,
    all_models=False,
    all_chains=True,
    in_place=True
    )
```

This function gets a (list of) structures, from a PDB file, and modify the b-factor values of the CYS and PRO atoms to separate them in CYO / CYR and PRC / PRT. The values assigned in here are -1 / +1, but these are selected rather arbitrarily (without any physical meaning).

:param structure: (list) of models directly from a PDB file.

:param all_models: (bool) flag to indicate whether we want to process all models in the list, or only the first model. Default is “False”.

:param all_chains: (bool) flag to indicate whether we want to process all chains, or only the first. Default is “True”.

:param in_place: (bool) flag to indicate whether we want to modify the input structure itself, or return a copy of it.

:return: the reference of the structure, with the correct modifications in the b-factors of CYS/PRO residues.

### Function `read_random_coil_to_df`

```
    def read_random_coil_to_df(
        f_path
    )
```

Read the Random Coil file and return the data in a DataFrame. The input file is generated from the PROSECCO program and has specific structure. E.g.:

`[ '#' 'ID' 'RES' 'Q3' 'CA' 'CB' 'C' 'H' 'HA' 'N' ]`

:param f_path: Input file with the random coil chemical shifts.

:return: A dataframe (pandas) with the data from the file.

## Classes

### Class `ChemShifts`

```
    class ChemShifts(
        N,
        C,
        CA,
        CB,
        H,
        HA
    )
```

ChemShifts(N, C, CA, CB, H, HA)

#### Ancestors (in MRO)

- builtins.tuple

#### Instance variables

**Variable `C`** Random coil chemical shift value for ‘C’.

**Variable `CA`** Random coil chemical shift value for ‘CA’.

**Variable `CB`** Random coil chemical shift value for 'CB'.

**Variable `H`** Random coil chemical shift value for 'H'.

**Variable `HA`** Random coil chemical shift value for 'HA'.

**Variable `N`** Random coil chemical shift value for 'N'.

## Module `Napshift.src.chemical_shifts.hydrogen_bond`

This module includes the main class that handles the calculation of the hydrogen bonds.

### Classes

#### Class `BondInfo`

```
    class BondInfo(
        res_name,
        res_id,
        H_,
        O_
    )
```

BondInfo(res_name, res_id, H_, O_)

##### Ancestors (in MRO)

- builtins.tuple

##### Instance variables

**Variable `H_`** Information about the NH-bond: (distance, cos(phi), cos(psi), 1.0).

**Variable `O_`** Information about the CO-bond: (distance, cos(phi), cos(psi), 1.0).

**Variable `res_id`** Residue ID (from the PDB file).

**Variable `res_name`** Residue name (three letter code) that contains the hydrogen bond.

#### Class `HBond`

```
    class HBond(
        verbose=False
    )
```

This class creates an object that handles hydrogen bonds formation between atoms (amino-acid chains).

`Note: Naming systems used (BMRB / PDB).`

Constructs an object that will handle the detections of hydrogen bonds in residue-chains.

:param verbose: boolean flag. If true it will display more information while running.

### Instance variables

**Variable `back_H`** Return an attribute of instance, which is of type owner.

**Variable `rule`** Return an attribute of instance, which is of type owner.

**Variable `rules`** Accessor of the hydrogen bond rules.

:return: Dictionary with the current distances and angles.

**Variable `side_H`** Return an attribute of instance, which is of type owner.

**Variable `side_O`** Return an attribute of instance, which is of type owner.

**Variable `verbose`** Accessor (getter) of the verbose flag.

:return: _verbose.

### Methods

#### Method `update_rule`

```
def update_rule(
    self,
    new_value,
    code_str=None
)
```

Update the values of the hydrogen bond rules.

:param new_value: distance (Angstrom) or angle (degrees).

:param code_str: Rule codes are ["D-A", "H-A", "PHI", "PSI"].

:return: None.

# Module `Napshift.src.chemical_shifts.input_vector`

This module includes the main class that handles the data generation from the (input) PDB files.

## Classes

### Class `InputVector`

```
class InputVector(
    blosum_id=62,
    include_hydrogen_bonds=False,
    check_aromatic_rings=True,
    data_type=numpy.float16
)
```

This class creates an input vector for the artificial neural network. The hydrogen bonds, and aromatic rings calculations are performed by class objects. We usually don't want to change these often once they are fixed.

Constructs an object that will create the input vectors from a given PDB (protein) file.

:param blosum_id: (integer) value that defines the version of the block substitution matrix.

:param include_hydrogen_bonds: (boolean) flag. If true it will detect and include all the hydrogen bonds information as part of the input vectors.

:param check_aromatic_rings: (boolean) flag. If true it will detect the target atoms that are affected by an aromatic ring. This will flag them out so that we will not use them while training/testing of the ANN.

:param data_type: This is the datatype of the input vector. This will affect the size (unit is MB) of the return vectors. Default here is set to float16.

### Class variables

**Variable `aromatic_ring_calc`** The default distance threshold is set equal to 3 Angstrom. It is also enabled to include five atom rings (default).

**Variable `hydrogen_bond_calc`** The default setting for the hydrogen bond calculator are:

```
> "D-A": 3.90, "H-A": 2.50
> "PHI": 90.0, "PSI": 90.0
```

Note: The distance units are in "Angstrom" and the angle units are in "degrees".

### Instance variables

**Variable `aromatic_rings`** Return an attribute of instance, which is of type owner.

**Variable `blosum`** Return an attribute of instance, which is of type owner.

**Variable `blosum_id`** Return an attribute of instance, which is of type owner.

**Variable `blosum_version`** Accessor (getter) of the version of the BLOSUM.

:return: integer value.

**Variable `check_aromatic_rings`** Accessor (getter) of the boolean flag that indicates if we detect the aromatic rings.

:return: boolean flag.

**Variable `data_type`** Return an attribute of instance, which is of type owner.

**Variable `hydrogen_bonds`** Return an attribute of instance, which is of type owner.

**Variable `include_hydrogen_bonds`** Accessor (getter) of the boolean flag that indicates if we include the hydrogen bonds in the formulation of the input vector.

:return: boolean flag.

### Static methods

#### Method `save_auxiliary`

```
def save_auxiliary(
    f_id,
    rec_data,
    kind=None,
    output=None
)
```

This auxiliary (static) function will save the auxiliary bi-products of the input vector construction, such as the hydrogen bonds, torsion angles and aromatic rings.

:param f_id: File id. Usually the PDB-ID is ok to be used as a filename to identify the contents.

:param rec_data: The data we want to save. Usually it is a list of things (tuples, dicts, etc.)

:param kind: This is the type / kind of data that we are saving. It can only be of four types: 1) “t_peptides”, 2) “h_bonds”, 3) “a_rings” and 4) “t_angles”. Anything else will force the method to raise an exception.

:param output: This is the main (parent) output directory where the data will be saved.

:return: None.

#### Method `sine_cosine`

```
def sine_cosine(
    angle
)
```

Auxiliary function that returns the sin(x) / cos(x) for a given input angle ‘x’. This method is using Numba for speed up. Upon testing the performance we found that we average an ~23x speed up comparing to using only numpy.

NOTE: We don’t make any checks on the input angle as we assume that it has already been checked before this call. The whole point is to speed up the repeated calls to the trigonometric functions.

:param angle: the angle (in degrees) that we want to get the sine / cosine.

:return: sin(angle), cos(angle)

### Methods

#### Method `get_data`

```
def get_data(
    self,
    f_path,
    n_peptides=3,
    all_models=False,
    save_output_path=None,
    verbose=False
)
```

This is the main function of the ‘InputVector’ class. It accepts as input a “Path(/path/to/the/PDB)” and returns a list with all the information that is extracted. This can be used as input to the ANN for predicting the chemical shift values of specific pre determined, atoms (e.g. “N”, “C”, “CA”, “CB”, “H”, “HA”).

:param f_path: Path of the (protein) PDB file.

:param n_peptides: Number of peptides to consider for the input vectors. By default it considers tri-peptides.

:param all_models: (bool) flag. If True the method will process all the models in the PDB file, otherwise only the first model.

:param save_output_path: (Path/String) If given it will be the output path where all the auxiliary data will be saved.

:param verbose: (bool) flag. If True it will print more info on the screen. The default is set to False.

:return: A dictionary with (data + sequence) for each processed model from the PDB file. The data + sequence are given as:

```
1) a list with the following information:
    1.1) poly-peptides that have been generated
         (index + three letter code)
    1.2) vector with the input values
    1.3) a list with the target atoms that are
         available for each vector.

2) the (modified) amino-acid sequence in string.
```

## Module `Napshift.src.chemical_shifts.model_machine`

This module includes the main classes that handle the training of the ANN models as well as the predictions of the new chemical shifts.

### Classes

#### Class `ChemShiftBase`

```
class ChemShiftBase(
    dir_input=None,
    dir_output=None,
    overwrite=True,
    file_logging=False
)
```

Constructs an object that holds the basic functionality of the rest classes. It mostly handles input/output directories along with some commonly used variables/flags.

:param dir_input: Input directory.

:param dir_output: Output directory.

:param overwrite: Overwrite file protection (flag). If is True then the output process WILL overwrite any pre-existing files.

:param file_logging: If the flag is True it will start logging the activities of the object in files.

##### Descendants

- Napshift.src.chemical_shifts.model_machine.ChemShiftPredictor
- Napshift.src.chemical_shifts.model_machine.ChemShiftTraining

##### Class variables

**Variable `dir_default`** The default directory is set to the “current working directory”.

### Instance variables

**Variable `dir_input`** Return an attribute of instance, which is of type owner.

**Variable `dir_output`** Return an attribute of instance, which is of type owner.

**Variable `input_path`** Accessor (getter) of the input path.

:return: dir_input.

**Variable `logger`** Return an attribute of instance, which is of type owner.

**Variable `output_path`** Accessor (getter) of the output path.

:return: dir_output.

**Variable `overwrite`** Accessor (getter) of the overwrite flag.

:return: overwrite_flag.

**Variable `overwrite_flag`** Return an attribute of instance, which is of type owner.

### Class `ChemShiftPredictor`

```
class ChemShiftPredictor(
    dir_model=None,
    dir_output=None,
    overwrite=True
)
```

Constructs an object that will perform the chemical shifts prediction. This is done by first constructing the necessary input values from the PDB file and subsequently calling the trained nn-models to perform the predictions on all atoms.

:param dir_model: (Path) Directory where the trained ANN models exist, (one for each target atom).

:param dir_output: (Path) Directory where the output of the predictions will be saved.

:param overwrite: (bool) Overwrite file protection. If “True”, then the output file WILL overwrite any pre-existing output.

### Ancestors (in MRO)

- Napshift.src.chemical_shifts.model_machine.ChemShiftBase

### Class variables

**Variable `df_avg`** This dataframe will be used in case we do not provide a random coil file.

**Variable `random_coil`** This object will be used to predict the random coil chemical shifts. The default value for the “pH” is set to ‘7.0’.

**Variable `vec_in`** This will be used to load the PDB file(s) before we make the prediction with the ANN model. Here we set (for brevity) all the input parameters.

### Instance variables

**Variable `input_scaler`** Return an attribute of instance, which is of type owner.

**Variable `nn_model`** Return an attribute of instance, which is of type owner.

### Methods

**Method `predict`**

```
def predict(
    self,
    f_path,
    n_peptides=3,
    all_models=False,
    random_coil_path=None,
    verbose=False,
    talos_fmt=True
)
```

Primary method of a "ChemShiftPredictor" object. It accepts a PDB file as input, constructs the input to the trained NN and puts the results -(predicted chemical shifts)- in a new text file.

:param f_path: (string) PDB file with the residue / atom coordinates.

:param n_peptides: (int) Number of peptides to consider for the input vectors. By default it considers tri-peptides.

:param all_models: (bool) flag. If "True" the method will process all the models in the PDB file, otherwise only the first model.

:param random_coil_path: (string) file with the random coil chemical shift values.

:param verbose: (bool) If "True" it will display more info during the prediction. The default set is "False" to avoid cluttering the screen with information.

:param talos_fmt: (bool) If "True" (default) it will use the TALOS format to save the results. If it is set to "False" the output format will be tabular.

:return: It will call the save method to write the results in a TALOS file format.

**Method `save_to_file`**

```
def save_to_file(
    self,
    pdb_id,
    aa_sequence,
    predictions,
    target_peptides,
    ref_peptides,
    random_coil=None,
    model_id=None,
    chain_id=None,
    talos_format=True
)
```

This method accepts the output of the **call**() method and writes all the information in a text file. Optionally we can save using the TALOS format.

:param pdb_id: (string) This is the PDB-ID from the input file. It is used to provide information in the final output file.

:param aa_sequence: (string) This is the sequence of the amino acids in the input file. It is used for information to the final output file.

:param predictions: These are the predictions (numerical values) of the ANN. It is a dictionary where each entry (key) corresponds to an atom-target.

:param target_peptides: These are the poly-peptides that were predicted by the artificial neural network. Because of the "aromatic-rings effect" there could be poly-peptides that were not predicted for all the targets.

:param ref_peptides: These are ALL the poly-peptides, as constructed by the InputVector class. They are used as reference with regards to the list of "poly_peptides".

:param random_coil: This is a DataFrame with the random coil values. If it isn't given (default=None) we will use average values from a default table.

:param model_id: This is the id of the model in the PDB file. We use it to distinguish the output results.

:param chain_id: This is the id of the chain in the protein model. We use it to distinguish the output results.

:param talos_format: (bool) Flag that defines the file format. If is set to "True" we will use the TALOS format. If it is set to "False" the file will be saved with a default tabular format.

:return: None.

### Class `ChemShiftTraining`

```
class ChemShiftTraining(
    dir_data=None,
    dir_output=None,
    overwrite=True
)
```

Constructs an object that will perform the training of the artificial neural networks (ANNs), on predicting the chemical shift values from specific atoms.

:param dir_data: Directory where the trained ANN models (one for each target atom) exist.

:param dir_output: Directory where the output of the training will be saved (trained ANN models).

:param overwrite: Overwrite file protection. If True, then the output process WILL overwrite any pre-existing output files.

#### Ancestors (in MRO)

- Napshift.src.chemical_shifts.model_machine.ChemShiftBase

#### Methods

##### Method `load_data`

```
def load_data(
    self,
    atom=None
)
```

This method will load the training datasets for a given input atom (target).

:param atom: Atom to be used as target during the training process by the ann.

:return: The (X/y) datasets. Note that the method will look for NaN target values and will clean up these values.

#### Method `train_models`

```
def train_models(
    self,
    validation_split=0.1,
    save_plots=True,
    verbose=False
)
```

The main purpose of this method is to use a pre-defined Artificial Neural Network and train it on the chemical shift data. Since we have six targets the method will train six networks separately and store its results.

:param validation_split: (float) This value is used to keep a portion of the training data aside while training to validate the training process. This is not the test set hold out.

:param save_plots: (bool) If "True" it will save the training errors, as function of time (epochs), for all the training targets (atoms).

:param verbose: (bool) If "True" it will display more information during the training. The default is "False" to avoid cluttering the screen with information.

:return: None.

## Module `Napshift.src.chemical_shifts.order_parameters`

This module includes the main class that handles the computation of the NH-order parameters.

### Classes

#### Class `NHOrderParameters`

```
class NHOrderParameters(
    r_eff=1.0
)
```

This class computes and returns the "S^2" NH order parameters, of a given input chain of amino-acids.

Constructs an object that will handle the calculation of order-parameters.

:param r_eff (float): Distance defines the interaction range of steric contacts [L: Ang].

##### Instance variables

**Variable `distance`** Accessor (getter) of the distance value.

:return: distance threshold value [L: Angstrom]

**Variable `r_eff`** Return an attribute of instance, which is of type owner.

## Module `Napshift.src.chemical_shifts.ring_effect`

This module includes the main class that handles the detection of the aromatic rings.

## Classes

### Class `RingEffects`

```
class RingEffects(
    d_ring=1.0,
    include_five_atoms=False,
    verbose=False
)
```

This class creates an object that will identify the rings (aromatic amino acids) and will also detect the atoms, from a given input chain, that are affected.

`Note: Naming systems used (BMRB / PDB).`

Constructs an object that will handle the detections of rings in the chain and the atoms that are affected by them.

:param d_ring: distance threshold [L: Angstrom].

:param include_five_atoms: (bool) flag for the Tryptophan amino-acid ring (with the five atoms).

:param verbose: (bool) if true it will display more information while running.

#### Instance variables

**Variable `d_ring`** Return an attribute of instance, which is of type owner.

**Variable `distance`** Accessor (getter) of the distance value that sets a threshold on the effect that the ring can have.

:return: distance threshold value [L: Angstrom]

**Variable `five_atoms_rings`** Return an attribute of instance, which is of type owner.

**Variable `include_five_atoms`** Accessor (getter) of the boolean flag that indicates if we consider both rings of the Tryptophan amino-acid.

:return: boolean flag.

**Variable `rings`** Return an attribute of instance, which is of type owner.

**Variable `verbose`** Accessor (getter) of the verbose flag.

:return: _verbose.

#### Methods

**Method `check_effect`**

```
def check_effect(
    self,
    chain=None,
    find_rings=True,
    exclude_self=True
)
```

Checks the input residue-chain to identify which atoms are affected by the aromatics rings.

:param chain: input amino-acid chain.

:param find_rings: if True, it will automatically find the rings of the same input chain.

:param exclude_self: if True, it will automatically exclude all the CB atoms of its own ring. These atoms are usually very close to the center of the ring.

:return: a list with atoms that are affected (given the threshold distance that we have already define).

#### Method `find_rings_in_chain`

```
def find_rings_in_chain(
    self,
    chain=None
)
```

Scan the input chain and look for aromatic rings. In the case of the Tryptophan, we can optionally consider both rings.

:param chain: input amino-acid chain.

:return: None.

#### Method `get_rings`

```
def get_rings(
    self
)
```

Accessor of the rings list.

:return: the list with the ring information.

### Class `RingInfo`

```
class RingInfo(
    res_name,
    res_id,
    coord
)
```

RingInfo(res_name, res_id, coord)

#### Ancestors (in MRO)

- builtins.tuple

#### Instance variables

**Variable `coord`** Coordinates (x,y,z) of the ring centroid.

**Variable `res_id`** Residue ID (from the PDB file).

**Variable `res_name`** Residue name (one letter code) that contains the aromatic ring.

## Module `Napshift.src.random_coil`

### Sub-modules

- Napshift.src.random__coil.camcoil
- Napshift.src.random__coil.random__coil__properties

## Module `Napshift.src.random_coil.camcoil`

This module provides a "Python implementation" of the camcoil program (originally written in C) to estimate the random coil chemical shift values from a sequence (string) of amino-acids.

The work is described in detail at:

1. Alfonso De Simone, Andrea Cavalli, Shang-Te Danny Hsu, Wim Vranken and Michele Vendruscolo (2009) (https://doi.org/10.1021/ja904937a). "Accurate Random Coil Chemical Shifts from an Analysis of Loop Regions in Native States of Proteins". Journal of the American Chemical Society (J.A.C.S.), 131 (45), 16332 - 16333.

Note ——= The txt files: 'corr__L1', 'corr__L2', 'corr__R1', 'corr__R2', are required for the estimation of the random coil values. They should be placed in the same directory with the module file. If they do not exist the code will exit with an error.

### Classes

#### Class `CamCoil`

```
class CamCoil(
    pH=7.0
)
```

This class implements the CamCoil code in Python.

Initializes the camcoil object. The pH is given as option during the initialization of the object even though only two actual implementations exist at the moment (i.e., pH=2 and pH=7).

If the user selects another pH value, this will be set automatically to one of these two in the code.

:param pH: (float) the default pH value is set to 7.0.

#### Instance variables

**Variable `df`** Return an attribute of instance, which is of type owner.

**Variable `pH`** Accessor (getter) of the pH parameter.

:return: the pH value.

#### Methods

#### Method `predict`

```
def predict(
    self,
    seq=None,
    verbose=False
)
```

Accepts a string amino-acid sequence, and returns a prediction with the random coil chemical shifts.

:param seq: (string) The input amino-acid sequence.

:param verbose: (bool) If the flag is set to True it will print more information on the screen.

:return: a pandas DataFrame, with the results.

## Module `Napshift.src.random_coil.random_coil_properties`

Properties for the CamCoil implementation. These include:

```
1. ACCEPTED_RES_ONE
2. pH2_prop
3. pH7_prop
4. weights
5. weights_LFP
```

## Module `Napshift.src.tests`

### Sub-modules

- Napshift.src.tests.test_auxiliries
- Napshift.src.tests.test_hbond
- Napshift.src.tests.test_input_vector
- Napshift.src.tests.test_ring_effect

## Module `Napshift.src.tests.test_auxiliries`

### Classes

#### Class `TestAuxiliaries`

```
class TestAuxiliaries(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'. * longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed. * maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

#### Ancestors (in MRO)

- unittest.case.TestCase

#### Static methods

##### Method `setUpClass`

```
def setUpClass() -> NoneType
```

Hook method for setting up class fixture before running tests in the class.

##### Method `tearDownClass`

```
def tearDownClass() -> NoneType
```

Hook method for deconstructing the class fixture after running all tests in the class.

#### Methods

##### Method `setUp`

```
def setUp(
    self
) -> NoneType
```

Creates the test object with default directory. This is set to the "current working directory".

:return: None.

##### Method `test_get_sequence`

```
def test_get_sequence(
    self
) -> NoneType
```

Test the get_sequence() function.

:return: None.

# Module `Napshift.src.tests.test_hbond`

## Classes

### Class `TestHBond`

```
class TestHBond(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named ‘runTest’.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test’s environment (‘fixture’) can be implemented by overriding the ‘setUp’ and ‘tearDown’ methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when the instance’s assertion methods fail; test methods raising this exception will be deemed to have ‘failed’ rather than ‘errored’. * longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed. * maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

#### Ancestors (in MRO)

- unittest.case.TestCase

#### Static methods

##### Method `setUpClass`

```
def setUpClass() -> NoneType
```

Hook method for setting up class fixture before running tests in the class.

##### Method `tearDownClass`

```
def tearDownClass() -> NoneType
```

Hook method for deconstructing the class fixture after running all tests in the class.

#### Methods

#### Method `setUp`

```
def setUp(
    self
) -> NoneType
```

Creates the test object with default parameters. :return: None.

#### Method `test_update_rule_angles`

```
def test_update_rule_angles(
    self
) -> NoneType
```

Test the “update_rule” method for angles. :return: None

#### Method `test_update_rule_distances`

```
def test_update_rule_distances(
    self
) -> NoneType
```

Test the “update_rule” method for distances. :return: None

## Module `Napshift.src.tests.test_input_vector`

### Classes

#### Class `TestInputVector`

```
class TestInputVector(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named ‘runTest’.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test’s environment (‘fixture’) can be implemented by overriding the ‘setUp’ and ‘tearDown’ methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when the instance’s assertion methods fail; test methods raising this exception will be deemed to have ‘failed’ rather than ‘errored’. * longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed. * maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

### Ancestors (in MRO)

- unittest.case.TestCase

### Static methods

#### Method `setUpClass`

```
def setUpClass() -> NoneType
```

Hook method for setting up class fixture before running tests in the class.

#### Method `tearDownClass`

```
def tearDownClass() -> NoneType
```

Hook method for deconstructing the class fixture after running all tests in the class.

### Methods

#### Method `setUp`

```
def setUp(
    self
) -> NoneType
```

Creates the test object with default setting.

:return: None.

#### Method `test_aromatic_rings_flag`

```
def test_aromatic_rings_flag(
    self
) -> NoneType
```

Test the aromatic rings flag get/set methods.

:return: None.

#### Method `test_blosum`

```
def test_blosum(
    self
) -> NoneType
```

Test the BLOSUM get/set methods.

:return: None.

#### Method `test_call_method`

```
def test_call_method(
    self
) -> NoneType
```

Test the call method of the class.

:return: None.

### Method `test_get_sin_cos_one`

```
def test_get_sin_cos_one(
    self
) -> NoneType
```

Test the sin_cos_one method with several angles.

:return: None.

### Method `test_hydrogen_bond_flag`

```
def test_hydrogen_bond_flag(
    self
) -> NoneType
```

Test the hydrogen bond flag get/set methods.

:return: None.

### Method `test_invalid_initializations`

```
def test_invalid_initializations(
    self
) -> NoneType
```

Test the constructor method for wrong initial values.

:return: None

# Module `Napshift.src.tests.test_ring_effect`

## Classes

### Class `TestRingEffects`

```
class TestRingEffects(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'. * longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed. * maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

#### Ancestors (in MRO)

- unittest.case.TestCase

#### Static methods

##### Method `setUpClass`

```
def setUpClass() -> NoneType
```

Hook method for setting up class fixture before running tests in the class.

##### Method `tearDownClass`

```
def tearDownClass() -> NoneType
```

Hook method for deconstructing the class fixture after running all tests in the class.

#### Methods

##### Method `setUp`

```
def setUp(
    self
) -> NoneType
```

Creates the test object with default parameters. :return: None.

##### Method `test_distance_methods`

```
def test_distance_methods(
    self
) -> NoneType
```

Test the distance accessor method. :return: None

##### Method `test_invalid_initializations`

```
def test_invalid_initializations(
    self
) -> NoneType
```

Test the constructor method for wrong initial values. :return: None

---

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).