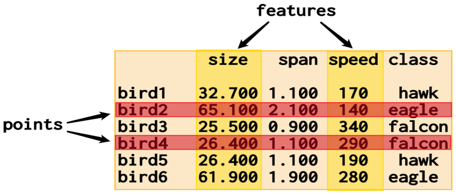
Nimble Cheatsheet

Nimble is built on top of Pythonâ€™s most popular data science and machine learning libraries to provide a single, easy to use, API for any data science job.

Nimble Data Object

Visualization of a Nimble data object:



	size	span	speed	class
bird1	32.700	1.100	170	hawk
bird2	65.100	2.100	140	eagle
bird3	25.500	0.900	340	falcon
bird4	26.400	1.100	290	falcon
bird5	26.400	1.100	190	hawk
bird6	61.900	1.900	280	eagle

Nimble has 4 data types that share the same API.

Each use a different backend to optimize the operations based on the type of data in the object. Choosing the type that best matches the data will support more efficient operations. By default, Nimble will attempt to automatically detect the best type.

Type	Data	Backend
List	**any data**	**Python list**
Matrix	**all the same type**	**NumPy array**
DataFrame	**each column has 1 type**	**Pandas DataFrame**
Sparse	**mostly missing or 0**	**SciPy coo_matrix**

A **Nimble data object** acts as the container of all individual elements of your data. But for manipulating that data, Nimble defines an API that abstracts away from the structure of how it is recorded to emphasize the meaning of how elements inter-relate.

Instead of operating on rows and columns (as with a spreadsheet or matrix), Nimble defines methods over **points** and **features**. This aligns with the goal of machine learning ready data, where each point should be a single observation of unique variables and each feature should define a single variable that has been recorded across observations. Nimble's API provides tools to tidy data towards that goal while behaving in a way that respects the observational meaning of data.

The methods of the Nimble data object control operations that apply to the entire object or the elements. The points and features properties of the object have additional methods for operations that apply along that axis of the data object.

```
bird1Size = X["bird1", "size"]              # Access an element by specifying a point and a feature
birdObs = X.points.copy(["bird1", "bird2"]) # Operates on whole points (seen as rows in image above)
labels = X.features.copy("class")           # Operates on whole features (seen as columns in image above)
duplicateObj = X.copy()                     # Operates on points and features simultaneously (whole object)
```

Note: Nimble also supports representations of higher-dimensional data by allowing multi-dimensional objects to be embedded within points. For example, each point could contain a two-dimensional image.

I/O

Creating Data

nimble.data is the primary function for loading data from all accepted sources. It accepts raw python objects, strings that are paths to files or urls, and open file objects.

```
X = nimble.data([[1, 'a'], [2, 'b']])
X = nimble.data('/path/to/X.csv')
```

For convenience, nimble.ones, nimble.zeros, and nimble.identity are available to quickly create objects with specific data. The following create objects with 10 points and 10 features.

```
allOnes = nimble.ones(10, 10)
allZeros = nimble.zeros(10, 10)
identity = nimble.identity(10)
```

nimble.random.data is available to construct an object of random data with adjustable sparsity. The following creates a Matrix object with 10 points, 10 features and 0 sparsity.

```
randomData = nimble.random.data(10, 10, 0, returnType='Matrix')
```

Fetching

Fetching returns the local path(s) to an online dataset, downloading and saving the data if necessary.

```
fileLocationsList = nimble.fetchFiles('UCI::iris')
```

Saving

Nimble data objectsÂ can be written to a file in a variety of formats. TrainedLearner objectsÂ can also be pickled.

```
X.save('data.csv')
X.save('data.pickle')
trainedLearner.save('learner.pickle')
```

Information about the data

Some information is set automatically on creation. By default automatic detection of returnType, pointNames, and featureNames occurs. Data object information can also be controlled by some of the parameters for nimble.data.

```
>>> X = nimble.data('/path/to/X.csv')
>>> X.shape                      # Always set
(3, 4)
>>> X.path                       # Set when source is a path
'/path/to/X.csv'
>>> X.getTypeString()            # Automatically detected
'Matrix'
>>> X.features.getNames()        # Automatically detected
['h', 'w', 'd']
>>> X.points.getNames()          # Automatically detected
['0k1r3', '6t3n1', '8i7i3', '0k2r2']
>>> headers = ['height', 'width', 'depth']
>>> items = ['couch', 'table', 'chair', 'love seat']
>>> X = nimble.data('/path/to/dataset.csv',
...                 pointNames=items', featureNames=headers,
...                 returnType="DataFrame",
name='furniture')
```

Once the object is created, the object's methods can be used to get or set information about the object.

```
X.name                           # A getter and setter
X.[points/features].getNames()
X.[points/features].getName(index)
X.[points/features].setNames(assignments)
X.[points/features].getIndex(identifier)
X.[points/features].getIndices(names)
X.[points/features].hasName(name)
len(X.[points/features])
```

Visualization

Printing

Nimble provides several ways to print or stringify the data, with varying levels of flexibility.

```
X                        # A representation of the data object that conforms to Python's repr standards
print(X)                 # A pretty-printed represenation of the data object
X.show(description, ...) # Pretty-print the object with customized parameters
```

Plotting

Nimble provides basic plotting functions using the matplotlib package on the backend.

```
X.plotFeatureAgainstFeature(x, y, ...)               # A scatter plot showing feature x plotted against feature y
X.plotFeatureAgainstFeatureRollingAverage(x, y, ...) # A rolling average of one feature plotted against another feature
X.plotFeatureDistribution(feature, ...)              # Plot a histogram of the distribution of values in a feature
X.plotFeatureGroupMeans(feature, groupFeature, ...)  # Plot the means of a feature grouped by another feature
X.plotFeatureGroupStatistics(statistic, feature,     # Plot an aggregate statistic for each group of a feature
                             groupFeature, ...)
X.plotHeatMap(...)                                   # Display a heat map of the data
X.[points/features].plot(identifiers, ...)           # Bar chart comparing points/features
X.[points/features].plotMeans(identifiers, ...)      # Plot means with 95% confidence interval bars
X.[points/features].plotStatistics(statistic,        # Bar chart comparing an aggregate statistic between points or
                                   identifiers, ...)   features
```

Iteration

Iteration can occur over elements, points, or features.

```
>>> for element in X.iterateElements(order, only):
...     print(element) # A single value
>>> for point in X.points:
...     print(point)   # A new Nimble data object containing the data from a single point
>>> for feature in X.features:
...     print(feature) # A new Nimble data object containing the data from a single feature
```

Querying

Data Querying

Many methods provide information about the data within a Nimble data object. The following functions provide information or perform calculations on the data, but they do not modify the data in the object or return a new Nimble data object.

```
X.countElements(condition)                    # The number of elements satisfying the query
X.countUniqueElements(...)                    # Values and counts of unique elements
X.containsZero()                              # True if any elements are equal to zero, otherwise False
X.report()                                    # Information describing the contents of the object
X.[points/features].count(condition)          # Number of points/features satisfying the query
X.[points/features].matching(function)        # Identify points/features satisfying the query
X.[points/features].similarities(function)    # Similarity calculations on each point/feature
X.[points/features].statistics(function, ...) # Statistics calculations on each point/feature
X.[points/features].unique()                  # Removal of duplicate points/features
X.features.report(basicStatistics,            # Statistical information about each feature
                  extraStatisticFunctions)
```

Indexing

Nimble uses **INCLUSIVE** indexes to support consistent behavior when using names or indices as identifiers.

Indexing can be performed from the data object or the points and features attributes.

```
data['bird2', 'speed']
data[1, 2]
data['bird2':'bird4', [0, 2]]
X.features["span"]
X.features[2]
X.points['bird4']
X.points[3]
X.features[:'speed']
X.points[3:]
```

Query Strings

For convenience, simple functions can be represented with strings. The strings must include a comparison operator (==, !=, >, <, >=, <=) or "is". An "is" (or "is not") must be followed by a nimble.match function or Python True, False, or None. See QueryString.

Element Query

```
numGreaterThan10 = X.countElements("> 10")
numNonMissing = X.countElements("is not missing")
```

Axis Query (using feature names from the example)

```
bigSpan = X.points.count("span > 30")
eagles = X.points.extract("class == eagle")
fast = X.points.copy("speed > 200")
```

Math

Operators

Python operators can be used between a Nimble data object and a scalar or two NimbleÂ data objects. The objects must be the same shape for elementwise operations and compatible shapes for matrix multiplication.

```
X + Y  # Elementwise Addition
X - Y  # Elementwise Subtraction
X * Y  # Elementwise Multiplication
X / Y  # Elementwise Division
X ** Y # Elementwise Power
X % Y  # Elementwise Modulo
X @ Y  # Matrix Multiplication
```

Stretch

The stretch property allows for expanded (broadcasting) computation with one-dimensional data objects. The one-dimensional object is stretched (repeated) to match the shape of the other object.

```
X + Y.stretch           # 2D + 1D
X.stretch / Y           # 1D / 2D
X.stretch * Y.stretch   # 1D * 1D
```

Linear Algebra

Linear algebra functions can also be applied to Nimble data objects.

```
X.matrixMultiply(other) # (same as using @ operator)
X.matrixPower(power)    # A square matrix raised to 'power' power
X.inverse()             # The inverse of the matrix
X.solveLinearSystem(b)  # Find the solution to a linear system
X.T                     # Returns the transposed object
```

Statistics

Methods

Bound methods for easy access to simple statistics, returning values for each point or feature packed into a new object.

```
X.[points/features].max(order)                         # Maximum values along the calling axis.
X.[points/features].mean(order)                        # Mean values along the calling axis.
X.[points/features].median(order)                      # Median values along the calling axis.
X.[points/features].medianAbsoluteDeviation(order)     # Median absolute deviations along the calling axis.
X.[points/features].min(order)                         # Minimum values along the calling axis.
X.[points/features].mode(order)                        # Modes along the calling axis.
X.[points/features].populationStandardDeviation(order) # Population standard deviations along the calling axis.
X.[points/features].proportionMissing(order)           # Proportions of values that are None or NaN along the calling axis.
X.[points/features].proportionZero(order)              # Proportions of values equal to zero along the calling axis.
X.[points/features].quartiles(order)                   # Quartiles along the calling axis
X.[points/features].standardDeviation(order)           # Standard deviations along the calling axis.
X.[points/features].uniqueCount(order)                 # Numbers of unique values along the calling axis.
X.[points/features].variance(order)                    # Variances along the calling axis.
```

By Axis vs By Object

The listed methods may also be called directly from an object, which will calculate against **ALL** values in the data, and then return that as a number. Mostly useful for objects which are themselves a single point or single feature.

```
>>> obj = nimble.data([0,2,4])
>>> featureMins = obj.features.min() # From axis
>>> print(featureMins)
1pt x 3ft
      0  1  2
    â”Œâ”€â”€â”€â”€â”€â”€â”€â”€
min â”‚ 0  2  4
>>> objectMin = obj.min(order)()     # From object
>>> print(objectMin)
0
```

Data Manipulation

Copying and Reordering

```
  X.copy(to)                                         # Make a deep copy of the object, optionally as a different object type
  X.[points/features].copy(toCopy, ...)              # Copy the points/features meeting a given criteria
† X.[points/features].permute(order)                 # Reorganize the points/features to be in a different order
† X.[points/features].sort(by, ...)                  # Sort the data based on point/feature values
```

Element Modification

```
† X.replaceFeatureWithBinaryFeatures(featureToReplace)          # Replace a categorical feature with one-hot encoded features
† X.replaceRectangle(replaceWith, pointStart, featureStart, ...) # Replace a section of the data with other data
† X.transformElements(toTransform, ...)                         # Change elements to new values
  X.calculateOnElements(toCalculate, ...)                       # Apply a calculation to each element
† X.transformFeatureToIntegers(featureToConvert)                # Map unique values to an integer and replace each element with the
                                                                  integer value
† X.[points/features].fillMatching(fillWith, matchingElements, ...) # Replace elements in points/features with a different value(s)
† X.[points/features].replace(data, ...)                        # Replace points/features with a new points/features
† X.[points/features].transform(function, ...)                  # Modify the elements within points/features
  X.[points/features].calculate(function, ...)                  # Apply a calculation to the elements within points/features
† X.features.normalize(function, ...)                           # Apply provided normalization function to features (optionally apply
                                                                  same normalization to the features of a second object)
```

Structural Changes

```
† X.transpose()                                      # Invert the points and features of this object (inplace)
† X.flatten(order, ...)                              # Deconstruct this data into a single point
† X.unflatten(dataDimensions, order, ...)            # Expand a one-dimensional object into a new shape
  X.groupByFeature(by, ...)                          # Separate the data into groups based on the value in a single feature
  X.trainAndTestSets(testFraction, ...)              # Separate the data into a training set and a testing set
† X.[points/features].append(toAppend)               # Add additional points/features to the end of the object
† X.[points/features].insert(insertBefore, toInsert, ...) # Add additional points/features at a given index
† X.[points/features].extract(toExtract, ...)        # Remove points/features from the object and place them in a new object
† X.[points/features].delete(toDelete, ...)          # Remove points/features from the object
† X.[points/features].retain(toRetain, ...)          # Keep certain points/features of the object
  X.points.mapReduce(mapper, reducer)                # Apply a mapper and reducer function to each point/feature
  X.[points/features].repeat(totalCopies, copyOneByOne) # Make a repeated copies of the object
```

† indicates an in-place operation that modifies the original data object rather than returning a copy

Helper Modules

nimble.calculate - Common calculation functions such as statistics and performance functions.

nimble.match - Common functions for determining if data satisfies a certain condition.

nimble.fill - Common functions for replacing missing data with another value.

nimble.random - Support for random data and randomness control within Nimble.

nimble.learners - Nimble's prebuilt custom learner algorithms.

nimble.exceptions - Nimble's custom exceptions types.

Machine Learning

Interfaces

Nimble interfaces with popular machine learning packages, to apply their algorithms within our API. Interfaces are used by providing "package.learnerName". For example:

```
nimble.train("nimble.RidgeRegression", ...)
nimble.trainAndApply("sklearn.KNeighborsClassifier", ...)
nimble.trainAndTest("keras.Sequential", ...)
```

The interfaces and learners available to Nimble are dependent on the packages installed in the current environment.

```
nimble.showAvailablePackages()
nimble.learnerNames()
nimble.showLearnerNames()
```

Learner Arguments

Find the parameters and any default values for a learner.

```
nimble.learnerParameters(name)              # A list of parameters that the learner accepts
nimble.showLearnerParameters(name)          # Print parameters of the learner
nimble.learnerParameterDefaults(name)       # A dict of parameters to their default values
nimble.showLearnerParameterDefaults(name)   # Print the default values of the learner
```

Trained Learner

The nimble.train function returns a TrainedLearner (referred to as "tl" below").

```
tl.learnerName                            # The name of learner used for training
tl.arguments                              # The arguments used for training
tl.randomSeed                             # The randomSeed applied for training
tl.tuning                                 # Tuning object containing the
                                            hyperparameter tuning results
tl.apply(testX, ...)                      # Apply the trained learner to new data
                                            data
tl.getAttributes()                        # Dictionary with attributes generated by the
                                            learner
tl.getScores(testX, ...)                  # The scores for all labels for each data point
tl.incrementalTrain(trainX, trainY, ...)  # Continue to train with additional data
tl.retrain(trainX, trainY, ...)           # Train the learner again on different data
tl.save(outPath)                          # Save the learner for future use.
tl.test(performanceFunction,              # Evaluate the accuracy of the learner on
        testX, testY, ...)                  testing data
```

Training, Applying, and Testing

The same API is available for any available learner.

```
trainedLearner = nimble.train(learnerName, trainX, trainY, ...)                              # Learn from the training data. Returns a TrainedLearner
predictedY = nimble.trainAndApply(learnerName, trainX, trainY, testX, ...)                   # Make predictions on new data
performance = nimble.trainAndTest(learnerName, performanceFunction, trainX, trainY, testX, testY, ...) # Evaluate the accuracy of the predictions on the testing data
performance = nimble.trainAndTestOnTrainingData(learnerName, performanceFunction, trainX, trainY, ...) # Evaluate the accuracy of the predictions on the used for training
normalizedX = nimble.normalizeData(learnerName, trainX, ...)                                 # Transform the training (and optionally testing) data using the learnerName
                                                                                               specified normalization
filledX = nimble.fillMatching(learnerName, matchingElements, trainX, ...)                    # Replace matching elements in points/features with provided or calculated values
```

Arguments can be set in two ways: by using the arguments parameter in the nimble function or by passing the learner object's parameters as keyword arguments. Hyperparameter tuning is triggered by annotating the parameters in question with a nimble.Tune object. and by passing a nimble.Tuning object into training.

```
>>> tl = nimble.train("sklearn.KNeighborsClassifier", trainX, trainY, arguments={'n_neighbors': 7})
>>> tl = nimble.train("sklearn.KMeans', trainX, trainY, n_clusters=7)
>>> tuningObj = nimble.Tuning(validation=0.2, performanceFunction=rootMeanSquareError)
>>> tl = nimble.train("sklearn.Ridge', trainX, trainY, alpha=nimble.Tune([0.1, 1.0]), tuning=tuningObj)
```