Data Analysis Made Simple with Pharo DataFrame

Oleksandr Zaitsev, Cyril Ferlicot-Delbecque

March 12, 2024

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Introduction

Data frames are one of the essential parts of the data science toolkit. They are the specialized data structures for tabular data sets that provide us with a simple and powerful API for summarizing, cleaning, and manipulating a wealth of data sources that are currently cumbersome to use.

A data frame is like a database inside a variable. It is an object which can be created, modified, copied, serialized, debugged, inspected, and garbage collected. It allows you to communicate with your data quickly and effortlessly, using just a few lines of code. The DataFrame project is similar to the *pandas*library in Python or the built-in *data.frame*class in R.

This booklet serves as the main source of documentation for the DataFrame project in Pharo. We'll start by describing the complete API of the DataFrame and DataSeries data structures, providing examples for each method. Then we'll show how data frames can be used in practice in a series of hands-on tutorials.

1.1 The DataFrame project

DataFrame v1.0 was the first implementation of data frames in Pharo. It was created by Oleksandr Zaitsev in 2017 as a Google Summer of Code project and was presented at the 25th International Smalltalk Joint Conference (ESUG 2017) in Maribor, Slovenia.

Thanks to the questions, requests, and suggestions of numerous people who were using DataFrame, the API was improved, some major bugs were fixed, several methods were aggressively optimized, and new features were added. Now with this booklet, we are happy to present the cleaned, improved, and updated DataFrame v2.0.

1.2 Who is this booklet for?

DataFrame was designed for anyone working with tabular data in Pharo. Being designed by a data scientist, the emphasis is on the typical data analysis that is commonly performed in a data science or machine learning workflow. However, knowledge of machine learning or statistics is not required to read this booklet or work with DataFrame.

On the other hand, familiarity with the Pharo programming language and environment is a prerequisite. In the next section, we'll briefly introduce Pharo. But it is also strongly recommended to learn Pharo and make sure that you are comfortable with it before you proceed to the following chapters.

DataFrame was designed in the tradition of Smalltalk and with *smalltalkers* in mind. If you come from a Python, R, or SQL background, some elements of the API might seem strange to you. But stick around and you will come to love the Smalltalk ways.

1.3 What is Pharo?

Pharo is a dynamic, purely object-oriented programming language (everything is an object) in the tradition of Smalltalk. But it is also a powerful development environment, focused on simplicity and immediate feedback. Its entire syntax fits on a postcard, and coding can be done directly in the debugger. Pharo has super cool tools that empower you and make you highly efficient.

Pharo's goal is to deliver a clean, innovative, free and open-source immersive environment. By providing a stable and small core system, excellent development tools, and maintained releases, Pharo is an attractive platform to build and deploy mission-critical applications.

Pharo fosters a healthy ecosystem developed by both private and commercial contributors who advance and maintain the core system and its external packages. More information about Pharo is available at http://www.pharo.org/. If you are new to Pharo, we suggest you take the Pharo MOOCor read the Pharo by Examplebook.

DataFrame v2.0 was tested on stable versions of Pharo 6.1 and Pharo 7.0, as well as on the development version of Pharo 8. Through continuous integration (CI) and 90% code coverage by unit tests, we try to make sure that DataFrame is working well on every latest stable and development version of Pharo on all major operating systems: Linux, Mac OS, and Windows, as well as both 32-bit and 64-bit architectures.

So before moving on to the next chapter, go to https://pharo.org/download and get yourself the latest Pharo. We strongly suggest that you try out all the examples as you read this booklet. It will help you understand what data frames are and how they can be used.

1.4 The PolyMath project

DataFrame was originally created as part of the PolyMath project - an open-source initiative that brings the power of scientific computing to Pharo. It started with the creation of the *PolyMath*library - a general purpose numerical computation framework in Pharo, similar to the *numpy*and *scipy*libraries in Python, as well as the *SciRuby*library and the built-in functionality of R and Julia.

PolyMath provides two main data structures: `PMMatrix` and `PMVector`, and covers topics such as statistical moments, polynomials, interpolations, integration, series (not to be confused with the data series that will be discussed in the following chapters), vector algebra, ordinary differential equations, complex numbers, quaternions, KD-trees, random number generation, arbitrary floating-point arithmetics, etc.

The original version of PolyMath was described in the book *Numerical methods in Smalltalk* by Didier Besset. Since then, PolyMath has changed a lot, and now, thanks to many different contributors, it has grown into a fully functional framework for numerical computations.

Besides PolyMath itself, the PolyMath organization on GitHub hosts other projects, including DataFrame, described in this booklet, and *libtensorflow-pharo-bindings*– a Pharo bridge to Google's *TensorFlow*library.

1.5 Terminology

At this point, you have probably noticed that, depending on the context, the phrase ”data frame” can refer to several different things. It can be a data structure, a class, a package, or this whole project. To remove these ambiguities from our language, we propose the following terminology and stick to it in this booklet:

data frame – a general reference to the tabular data structure (collection) designed for data analysis.

data series – a general reference to a dictionary-like data structure that is used to represent rows and columns of a data frame.

DataFrame – the class name of a data structure that models a data frame, or a reference to the DataFrame package or project.

DataSeries – the class name of a data structure used to model a data series.

We would also like to remind you that *"series"* is both a singular and a plural form. We say *"a single data series was created"* and *"two data series were removed"*.

1.6 Structure of this booklet

The rest of this booklet is structured in the following way:

- **Chapter 1. DataFrame by example** - this chapter will guide you through the complete API of the DataFrame project using simple examples.
- **Chapter 2. Tutorials** - a series of short tutorials that will teach you to work with data frames and demonstrate how they can be applied to practical problems.

1.7 How to contribute?

DataFrame is an open-source project released under the MIT public license. Although the original codebase was written by Oleksandr Zaitsev, the DataFrame project is the collaborative effort of many people who helped by giving advice, sharing ideas, reporting issues, and suggesting how to resolve them. There are many ways in which you can help make DataFrame better.

If you want to implement new features or improve existing functionality, you can fork the DataFrame repository on GitHub, make your changes, and create a pull request.

But contributions can be more simple and less time consuming. You can report a bug, suggest a better API, or request new features for DataFrame. All that can be done by creating dedicated issues in DataFrame's repository on GitHub.

Positive feedback is also very important. It is always very encouraging to hear how people use your project. So if you want to share your experience working or just playing with DataFrame, you can write about it on social media using the `#Pharo` and `#PharoDataFrame` hashtags. Also, feel free to contact me by email at *olk.zaytsev@gmail.com*.

CHAPTER 2

DataFrame by example

In this chapter, we will guide you through the complete functionality of the DataFrame project by showing examples of how each feature can be used. This is the documentation of DataFrame written in a form of storytelling with data.

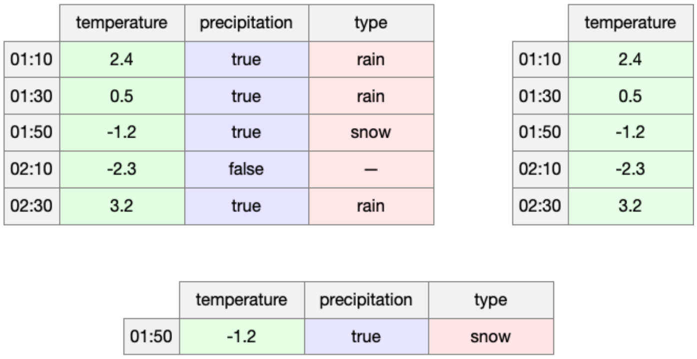
2.1 Weather dataset

For demonstrational purposes, we have designed a simple dataset of meteorological observations. Imagine that to study the weather in a certain area, you are collecting data by measuring the temperature and logging the type of precipitation (rain, snow etc.) every 20 minutes. The initial table contains only 5 observations (rows) and 3 features (columns): `temperature`, `precipitation`, and `type`. You can see this table in the top left of Figure 2-1. As we progress through this chapter, we will be modifying, adding, and removing rows and columns of this dataset. Two additional columns that we will add are `humidity` and `wind`.

2.2 What are data frames and data series

Before moving forward, you need to get a better understanding of the basic data structures that we use to model tabular data. In this section, we will explain the theoretical aspects of the data frame and data series collections.

Data frame is a table of data. Similar to an Excel spreadsheet or a relational database, but implemented as a collection that can be stored in a variable, such as an `Array` or `Dictionary`. This greatly simplifies the data analysis workflow. Because, to answer questions such as *"What was the temperature at*



	temperature	precipitation	type
01:10	2.4	true	rain
01:30	0.5	true	rain
01:50	-1.2	true	snow
02:10	-2.3	false	—
02:30	3.2	true	rain

	temperature
01:10	2.4
01:30	0.5
01:50	-1.2
02:10	-2.3
02:30	3.2

	temperature	precipitation	type
01:50	-1.2	true	snow

Figure 2-1 Weather data frame and two data series extracted from it: homogeneous series of temperature column and heterogeneous series of the third row

01:30?" or *"What is the average temperature when it snows?"*, we do not need to set up a whole database. Instead, we can simply load our data into a variable and query it - all in three lines of code (you will learn how to do it by reading the rest of this chapter):

```
weather := DataFrame readFromCsv: 'weather.csv'.

"Question 1"
weather at: 01:30 at: #temperature.

"Question 2"
(weather group: #temperature by: #type aggregateUsing: #average) at:
    #snow.
```

In the top left of Figure 2-1, you can see our weather dataset represented as a data frame. It contains three columns: a `Float` column of temperatures, a `Boolean` column which tells you whether or not there was any precipitation, and a `String` column that specifies the type of precipitation. Data types of the columns are represented with different background colors.

It is common to think of rows as observations and columns as features. Therefore, columns are usually homogeneous - they contain values of a same data type, and rows can be heterogeneous - containing values of different types, such as `Integer`, `Float`, `Boolean`, and `String`.

Each row or column can be extracted as a `DataSeries` object - a dictionary of key-value pairs with a name. The keys of a row are the column names and for a column there are the row names of the data frame. This means that, by having those keys and a name, every individual row or column contains all the information that is needed to identify its precise location in a data frame, as well as the meaning of each value. As a result, a data series is much more interpretable than a simple array of values, which allows us to construct more meaningful queries while analysing the dataset. We can add two series together, multiply them by a number, calculate the variance or standard deviation of a series, select elements that satisfy a certain condition, and do many other things that will be described in the following sections.

You can see an example of two data series in the top right and bottom of Figure 2-1. The `temperature` column is a homogeneous `Float` data series and the row `01:50` is a heterogeneous series containing different types of values.

In the following section we will show you how to create a weather data frame and analyse it in Pharo. But first you need to install DataFrame v2.0 and make sure that all its tests are passing.

2.3 Installation

To install DataFrame, go to the Playground (Ctrl+O+W) in your fresh Pharo image and execute the following Metacello script (select it and press the Do-it button or Ctrl+D):

```
Metacello new
  baseline: 'DataFrame';
  repository: 'github://PolyMathOrg/DataFrame/src';
  load.
```

For all keyboard shortcuts mentioned in this booklet the *Ctrl* key is for Windows and Linux. On Mac OS, use *Cmd* instead.

Running the tests

The first thing you should do after installing DataFrame is open the DataFrame-Tests package in the Test Runner (Ctrl+O+U) or System Browser (Ctrl+O+B) and make sure that all tests are passing. DataFrame v2.0 is tested with 301 unit tests which provide 90% code coverage. If you see some failing tests, please go to the DataFrame repository on GitHub and open a related issue.

2.4 Creating a data series

`DataSeries` behaves like an `OrderedDictionary`. You can create one by providing keys, values, and a name:

```
DataSeries
  withKeys: (#('01:10' '01:30' '01:50' '02:10' '02:30') collect:
    #asTime)
  values: #(2.4 0.5 -1.2 -2.3 3.2)
  name: #temperature.
```

This creates the first column of our weather data frame.

	temperature
1:10 am	2.4
1:30 am	0.5
1:50 am	-1.2
2:10 am	-2.3
2:30 am	3.2

If you don't specify a name, it will be set to the default value `'(no name)'`:

```
DataSeries
  withKeys: #(temperature precipitation type)
  values: #(0.5 true rain).
```

	(no name)
temperature	0.5
precipitation	true
type	rain

You can also create a data series without keys, in which case they will be filled with default values: numbers from 1 to the size of your data series:

```
temperature := DataSeries
  withValues: #(2.4 0.5 -1.2 -2.3 3.2)
  name: #temperature.
```

	temperature
1	2.4
2	0.5
3	-1.2
4	-2.3
5	3.2

Or only with values:

```
DataSeries
  withValues: #(2.4 0.5 -1.2 -2.3 3.2).
```

	(no name)
1	2.4
2	0.5
3	-1.2
4	-2.3
5	3.2

That last expression has a shorter form which produces the same result:

```
#(2.4 0.5 -1.2 -2.3 3.2) asDataSeries.
```

2.5 Discovering the data series API

`DataSeries` is an extended `OrderedDictionary`. It combines the API of both `Dictionary` and `SequenceableCollection`, and adds some additional functionality that can be useful for data analysis. This simple data structure does not require a long introduction but it is still worthwhile to demonstrate how it can be used. In this section, we briefly cover the public API of the `DataSeries` class and describe the additional methods that are not present in other Pharo collections.

Accessing and modifying values

In this aspect, `DataSeries` acts like an `OrderedDictionary`. You can access an element at a certain key using the `at: aKey` message, you can modify this element with `at: aKey put: newValue`. To remove an element, use `removeAt: aKey`. If `aKey` is not present in the data series, a new element will be created and added to the end. Alternatively, you can use methods such as `at: ifAbsent:`, `at: ifAbsentPut:`, etc. to provide a custom block that will be evaluated when `aKey` is not found.

You can also access elements by their index (rather than key) with the `atIndex:`, `atIndex: put:`, and `removeAtIndex:` methods.

Here is an incomplete list of accessors provided by `DataSeries`. Many others are inherited from the `OrderedCollection` class, which can come in handy but will not be discussed in this booklet:

```
DataSeries >> at:
DataSeries >> at: ifAbsent:
DataSeries >> at: put:
DataSeries >> removeAt:
DataSeries >> atIndex:
DataSeries >> atIndex: put:
DataSeries >> removeAtIndex:
```

Now we present the non-traditional and powerful API of `DataSeries`.

2.6 at:transform:

`DataSeries` provides an additional set of `at: transform:` modifying accessors. Consider a situation when you want to convert a certain value of the temperature data series from Celsius to Fahrenheit. Having to specify the key twice would make the whole expression overrly complex:

```
temperature at: '01:30' asTime put: (temperature at: '01:30' asTime)
     * 9/5 + 32.
```

And since operations on data series (rows and columns) are the building block of most data frame queries, we want them to be as short and readable as possible. To simplify complex queries, `DataSeries` provides a set of methods that allow you to transform an element at a certain position using a block:

```
DataSeries >> at: transform:
DataSeries >> at: transform: ifAbsent:
DataSeries >> atIndex: transform:
```

We can now rewrite the above expression as

```
temperature at: '01:30' asTime transform: [ :x | x * 9/5 + 32 ].
```

2.7 Enumerating the values of a data series

As a combination of `Dictionary` and `SequenceableCollection`, `DataSeries` understands the following methods for enumerating its values:

Value	**Value and key**	**Value and index**
do:	withKeyDo:	withIndexDo:
select:	withKeySelect:	withIndexSelect:
reject:	withKeyReject:	withIndexReject:
collect:	withKeyCollect:	withIndexCollect:
detect:	withKeyDetect:	withIndexDetect:
detect: ifNone:	withKeyDetect: ifNone:	withIndexDetect: ifNone:
inject: into:	-	-

For example, we can collect the Fahrenheit values of `temperature` into a separate series:

```
fahrenheit := temperature collect: [ :x | x * 9/5 + 32 ].
```

	temperature
1	36.32
2	32.9
3	29.84
4	27.86
5	37.76

Notice that the name of this data series is still `temperature`, because that was the name of the series from which the values were collected, and no other name was provided by the `collect:` message. If needed, you can rename the `fahrenheit` series later:

```
fahrenheit name: #fahrenheit.
```

To see another example of enumerating the values of a data series, let's now find the first value of the `fahrenheit` series that is below 32 (zero on Celsius scale):

```
fahrenheit detect: [ :x | x < 32 ].
```

The answer will be `29.84` - the third element of our series.

2.8 Arithmetical operations

Similar to other collections in Pharo, `DataSeries` responds to basic arithmetical operations, such as `+, -, *, /`. You can apply these operations on any of the following combinations: series-series, series-scalar, or scalar-series (although, the last one is not supported for division).

In order to demonstrate the application of arithmetical operations, let's create two simple data series filled with numbers:

```
a := DataSeries withValues: #(0.5 2 -1 0) name: #a.
b := DataSeries withValues: #(-2 -0.5 1 3) name: #b.
```

When an operation is applied to a scalar and a series, it is performed on every element of a series with the given scalar. For example, if you divide a data series `a` by `2`, each of its elements will be divided by `2`:

```
a / 2.
```

	a
1	0.25
2	1
3	-0.5
4	0

Similarly, if you subtract data series `b` from the number `1`, you will get a new series, in which each element is the difference between `1` and the corresponding element of series `b`:

```
1 - b.
```

	b
1	3
2	1.5
3	0
4	-2

When an operation is applied to a pair of data series, it will be performed in the elementwise manner. So, for example, if you want to add two data series `a` and `b`, you will get a third series where the first element is the sum of the first elements of `a` and `b`, the second element is the sum of the second elements, etc. In this case, both data series must be of the same size and must have the same keys. The name of the resulting series will be the same as the name of the first operand:

```
a + b.
```

	a
1	-1.5
2	1.5
3	0
4	3

In fact, the conversion of a `temperature` series to the Fahrenheit scale, which was demonstrated earlier in Section 2.7 with the `collect:` message, can be done directly by applying arithmetical operations to the data series:

```
fahrenheit := temperature * 9/5 + 32.
```

The result will be the same as before.

2.9 Some useful mathematical functions

Although you can apply any operation to the elements of your series using `collect:`, sometimes your code will be more readable if you send a message to the whole data series to be performed on each of its elements. In the same way as when we add 1 to a data series and get a new series with all the elements of the first series incremented by 1, we also want to say `series exp` and expect a data series of exponents.

Here is the list of elementwise mathematical operations that are understood by `DataSeries`:

```
DataSeries >> abs
DataSeries >> cos
DataSeries >> sin
DataSeries >> tan
DataSeries >> exp
DataSeries >> ln
DataSeries >> log
DataSeries >> log:
DataSeries >> sqrt
DataSeries >> **
```

The last one is the binary operator which takes a data series as the first operand and raises all its elements to the power of the second operand.

Let's find the exponent and the natural logarithm of the temperature series:

```
temperature exp.
```

	temperature
1	11.02
2	1.65
3	0.3
4	0.1
5	24.53

As we take the natural logarithm, the third and fourth elements in the resulting data series are `Float nan` because logarithms are only defined for positive numbers and those two temperatures are below zero:

```
temperature ln.
```

	temperature
1	0.88
2	-0.69
3	Float nan
4	Float nan
5	1.16

Unlike most other programming languages and libraries for numerical computations that define `log` as natural logarithm, Pharo defines it as a logarithm with base 10. Natural logarithm is defined as `ln` and the parametrised message `log:` can be used to get a logarithm with any other base, for example: `(1024 log: 2) = 10.0`.

2.10 Statistical operations

In addition to the statistical methods understood by all collections, such as `average`, `min`, or `max`, the `DataSeries` class provides more advanced methods that are commonly used when analysing quantitative columns. Here is the complete list of statistical methods understood by `DataSeries`:

```
DataSeries >> min
DataSeries >> max
DataSeries >> range
DataSeries >> average
DataSeries >> median
DataSeries >> mode
DataSeries >> quantile:
DataSeries >> quartile:
DataSeries >> zerothQuartile
DataSeries >> firstQuartile
DataSeries >> secondQuartile
DataSeries >> thirdQuartile
DataSeries >> fourthQuartile
DataSeries >> interquartileRange
```

If any of these methods is applied to a non-numerical data series, it will signal an exception.

To demonstrate the application of statistical methods, let's find the average, median, standard deviation, and variance of the temperature column:

```
temperature average. "0.52"
temperature median. "0.5"
temperature stdev. "2.3253"
temperature variance. "5.407"
```

We can also check if the following equalities hold: zeroth quartile should be the same as min, fourth quartile should be the same as max, second quartile - same as median, interquartile range equals third quartile minus the first one, range is max minus min, and variance is the square of the standard deviation:

```
temperature zerothQuartile = temperature min. "true"
temperature fourthQuartile = temperature max. "true"
temperature secondQuartile = temperature median. "true"
temperature interquartileRange = (temperature thirdQuartile -
    temperature firstQuartile). "true"
temperature range = (temperature max - temperature min). "true"
temperature variance = (temperature stdev ** 2). "true"
```

Summarizing a DataSeries

You can get a quick summary of the distribution of a numerical data series as a collection of its minimal and maximal values, first and third quartiles, average value, and a median.

Let's see how it works for the temperature series created in 2.4:

```
temperature summary.
```

	temperature
Min	-2.3
1st Qu.	-1.2
Median	0.5
Average	0.52
3rd Qu.	2.4
Max	3.2

2.11 Working with categorical values

Categorical data series have a fixed dictionary of values. For example, a series `sex` which has values `Male` and `Female`, a boolean series with values `true` and `false`, a series with sizes of clothes: `XS`, `S`, `M`, `L`, `XL`, etc. We can not perform mathematical and statistical operations on categorical series (except for mode, which only finds the most common value), however there is still a lot of ways to analyse such data. In the rest of this section, we will show you several methods of DataSeries that are especially useful for working with categorical values. The examples will be based on the precipitation and type columns of the weather dataset. Here is a reminder about how to create them:

```
keys := #('01:10' '01:30' '01:50' '02:10' '02:30') collect: #asTime.

precipitation := DataSeries
  withKeys: keys
  values: #(true true true false true)
  name: #precipitation.

type := DataSeries
  withKeys: keys
  values: #(rain rain snow - rain)
  name: #type.
```

removeDuplicates

The most basic operation that can be performed is removing the duplicate values of the series:

```
precipitation removeDuplicates. "#(false true)"
type removeDuplicates. "#(#snow #rain #-)"
```

In the past, this method was named #uniqueValues to follow the naming of Pandas. It got renamed to stay consistent with Pharo's API. This allows us to see the complete dictionary of values used in a data series.

valueCounts

Another thing you might want to know is how the values of your categorical data series are distributed. How many times did it rain, according to your dataset? How many times did it snow? The valueCounts method will give you the counts of unique values in your data series:

```
type valueCounts.
```

	type
rain	3
snow	1
-	1

The result will be a data series with unique values as keys and counts as values. It will be sorted by counts in descending order.

valueFrequencies

Sometimes, instead of counts, you want to see the relative frequencies of the unique values in your data series:

```
type valueFrequencies.
```

	type
rain	(3/5)
snow	(1/5)
-	(1/5)

In fact, it is the same as dividing value counts by the size of your series:

```
type valueFrequencies = (type valueCounts / type size). "true"
```

crossTabulateWith:

Cross tabulation is a powerful way to analyse the correlation between two categorical series. It creates a table with rows corresponding to the unique values of the first series, columns corresponding to the unique values of the second series, and each cell containing the count of those value pairs.

By cross tabulating the `precipitation` series with the `type` series, we can see that they are closely correlated - when `precipitation` is `false`, the `type` is always empty, and when `precipitation` is `true`, the `type` is either `rain` or `snow`:

```
precipitation crossTabulateWith: type.
```

The result will be the data frame of cross tabulated values:

	snow	rain	-
false	0	0	1
true	1	3	0

Categorising a DataSeries

It is often needed while manipulating numerical datas to categorise them.`DataSeries` api provides 2 methods to help with this:

- `#groupByBins:labelled:` : Return a new data series whose values will be a given label depending on the category found by the bins provided.
- `#groupByBins:` : Same as previous methods but using indexes starting at 1 as labels

Let's say for example that in our usecase, we do not need to have specific temperatures for our usecase but just some categorises. We can create a new data series like this:

```
temperature groupByBins: { Float negativeInfinity . 0 . 3 . Float
    infinity } labelled: #(#negative #very_cold #adequate).
```

The result will be a new `DataSeries` like this:

	temperature
1	#very_cold
2	#very_cold
3	#negative
4	#negative
5	#adequate

2.12 Handling nil values in a data series

The DataSeries class provides methods specifically for handling nil values in a data series. Consider this data series :

```
temperature := DataSeries
  withValues: #(2.4 nil -1.2 nil 3.2)
  name: #temperature.
```

Identifying nil values

The `hasNil` method returns true if the data series has at least one nil value.

```
temperature hasNil. "true"
```

Removing nil values

The `removeNils` method removes elements with nil values from the data series.

```
temperature removeNils.
```

key	**value**
1	2.4
3	-1.2
5	3.2

Replacing nil values

Rather than simply removing nil values from the data series, nil values can also be replaced by user defined or statistical alternatives.

- `replaceNilsWith: anObject` : Replaces all nil values in the data series with the provided object, `anObject`.
- `replaceNilsWithAverage` : Replaces all nil values in the data series with the average value of the data series.
- `replaceNilsWithMedian` : Replaces all nil values in the data series with the median of the data series.
- `replaceNilsWithMode` : Replaces all nil values in the data series with the mode of the data series.
- `replaceNilsWithNextValue` : Replaces all nil values in the data series with the value of the next non-nil element in the data series. If the last value in the data series is nil, it will remain nil even after using this method because there is no value after it which can replace it.
- `replaceNilsWithPreviousValue` : Replaces all nil values in the data series with the value of the previous non-nil element in the data series. If the first value in the data series is nil, it will remain nil even after using this method because there is no value before it which can replace it.
- `replaceNilsWithZero` : Replaces all nil values in the data series with zero.

Suppose the user wants to replace all the nil values with `5`.

```
temperature replaceNilsWith: 5.
```

key	value
1	2.4
2	5
3	-1.2
4	5
5	3.2

If you want to replace nil values with a statistical value such as the median of the data series :

```
temperature replaceNilsWithMedian.
```

key	value
1	2.4
2	2.4
3	-1.2
4	2.4

5	3.2

You can also replace nil values with adjacent values (the non-nil value appearing before the nil value in this example) in the data series :

```
temperature replaceNilsWithPreviousValue.
```

key	value
1	2.4
2	2.4
3	-1.2
4	-1.2
5	3.2

Counting nil values

You can count the number of nil values in a data series using `countNils` and the number of non-nil values in a data series using `countNonNils`.

```
temperature countNils. "2"
temperature countNonNils. "3"
```

2.13 Creating a data frame

In this section, we will look at different ways of creating the weather data frame described in Section 2.1. You will also learn to create empty data frames that can be filled with values later.

Initializing a data frame with an array of rows

The most basic way to initialize a data frame is with an array (or any other ordered collection) of rows where each row is a collection of elements. Let’s create the weather data frame from its rows:

```
weather := DataFrame withRows: #(
  (2.4 true rain)
  (0.5 true rain)
  (-1.2 true snow)
  (-2.3 false -)
  (3.2 true rain)).
```

Initializing a data frame with an array of columns

Alternatively, you can create a data frame by passing it a collection of columns. This can be handy, for example, when engineering new features: `DataFrame withColumns: { income . income ** 2 . income log }`. In our case, we create the same weather data frame:

```
weather := DataFrame withColumns: #(
  (2.4 0.5 -1.2 -2.3 3.2)
  (true true true false true)
  (rain rain snow - rain)).
```

Specifying column and row names

Both expressions in the two previous sections create the same data frame.

	1	**2**	**3**
1	2.4	true	rain
2	0.5	true	rain
3	-1.2	true	snow
4	-2.3	false	-
5	3.2	true	rain

Since the names of rows and columns were not specified, they were initialized with their default values: `(1 to: self numberOfRows)` and `(1 to: self numberOfColumns)`. We can provide more meaningful names:

```
weather := DataFrame withColumns: #(
  (2.4 0.5 -1.2 -2.3 3.2)
  (true true true false true)
  (rain rain snow - rain)).

weather columnNames: #(temperature precipitation type).
weather rowNames: (#('01:10' '01:30' '01:50' '02:10' '02:30')
    collect: #asTime).
```

We would like to emphasize that names don't have to be strings or numbers. They can be any objects, and in this case they are instances of Time. Now the data frame looks the same as the table in Figure 2-1 (A).

	temperature	**precipitation**	**type**
1:10 am	2.4	true	rain
1:30 am	0.5	true	rain
1:50 am	-1.2	true	snow
2:10 am	-2.3	false	-
2:30 am	3.2	true	rain

Compact methods for initializing data frames

The `DataFrame` class provides syntactic sugar that allows us to initialize it with contents and (optionally) row and column names in a single line. Here is the complete list of those initializers:

```
DataFrame class >> withRows: columnNames:
DataFrame class >> withRows: rowNames:
DataFrame class >> withRows: rowNames: columnNames:
DataFrame class >> withColumns: columnNames:
DataFrame class >> withColumns: rowNames:
DataFrame class >> withColumns: rowNames: columnNames:
```

Whenever row or column names are not specified, they are initialized with their default values.

2.14 Creating empty data frames

Sometimes we need to create an empty data frame that will be filled with data later on. The easiest way of doing this is with

```
DataFrame new. "empty data frame with 0 rows and 0 columns"
```

This will create an empty data frame with no columns and rows. You can also create an empty data frame of a given size by specifying it as a point `numberOfRows @ numberOfColumns`. All the cells of such a data frame will be empty (initialized with `nil`).

```
DataFrame new: 3@4. "empty data frame with 3 rows and 4 columns"
```

Similar to the syntactic sugar described in the previous section, the DataFrame class has methods that allow us to create data frames by specifying only their rows, columns, or both. Since we do not provide any data, such data frames will be empty and their sizes will correspond to the provided arrays of rows and columns. For example, if you create an empty data frame with columns `#(temperature precipitation type)`, its size will be `0@3`. Here are the methods that can be used to create an empty data frame with names:

```
DataFrame class >> withRowNames:
DataFrame class >> withColumnNames:
DataFrame class >> withRowNames: columnNames:
```

For example, we can initialize an empty weather data frame:

```
emptyWeather := DataFrame
  withRowNames: (#('01:10' '01:30' '01:50' '02:10' '02:30') collect:
    #asTime)
  columnNames: #(temperature precipitation type).
```

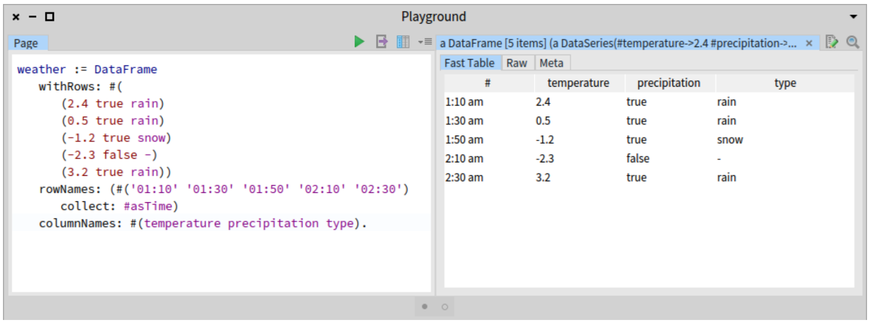
	temperature	**precipitation**	**type**
1:10 am	nil	nil	nil
1:30 am	nil	nil	nil

1:50 am	nil	nil	nil
2:10 am	nil	nil	nil
2:30 am	nil	nil	nil

Although it is possible to create a data frame with no rows using `DataFrame class >> new` or `DataFrame >> withColumnNames:` and add rows later by pushing them one by one, this is not recommended. DataFrame is not optimized for the insertion of new elements. It may be more efficient to first add rows into an OrderedCollection and then initialize a data frame with it.

2.15 Inspecting a data frame

It is hard to work with data tables without being able to see them. Fortunately, the Pharo environment provides a powerful inspector which allows you to examine live objects and even modify them. If you inspect a data frame object (select it anywhere in the code and press Ctrl+I or Ctrl+G if you are in Playground), you will see a table similar to the one in Figure 2-2.


Playground
Page
a DataFrame [5 items] (a DataSeries(#temperature->2.4 #precipitation->...
Fast Table
Raw
Meta

```
weather := DataFrame
   withRows: #(
      (2.4 true rain)
      (0.5 true rain)
      (-1.2 true snow)
      (-2.3 false -)
      (3.2 true rain))
   rowNames: (#('01:10' '01:30' '01:50' '02:10' '02:30')
      collect: #asTime)
   columnNames: #(temperature precipitation type).
```

#	temperature	precipitation	type
1:10 am	2.4	true	rain
1:30 am	0.5	true	rain
1:50 am	-1.2	true	snow
2:10 am	-2.3	false	-
2:30 am	3.2	true	rain

Figure 2-2 Inspecting a weather data frame

It is also possible to get info in a textual way by calling `#info` of the `DataFrame`:

```
weather info  "'DataFrame: 5 entries
Data columns (total 3 columns):
 # | Column | Non-nil count | Dtype
---------------------------------------------------
1 | 1 | 5 non-nil | SmallFloat64
2 | 2 | 5 non-nil | Boolean
3 | 3 | 5 non-nil | Object
'"
```

2.16 Data types

When importing datas, `DataFrame` will infer the types of the elements in the columns. In order to do that, all the elements of the colmun except nil values will be scanned to determine their type, and the common superclass of the elements will be selected as the type of the columns. If all elements are strings, this means `DataFrame` was not able to determine any type and the colmun will be classified as `Object`.

```
weather dataTypes  "a Dictionary(1->SmallFloat64 2->Boolean
    3->Object )"
```

It is possible to force a colmun type to be `String` be specifying it directly:

```
weather dataTypes at: 3 put: String.
weather dataTypes "a Dictionary(1->SmallFloat64 2->Boolean 3->String
    )"
```

2.17 Accessing data frame parameters

Data frames are defined by their contents (table of data), as well as their column and row names. For simplicity, we provide access to some additional parameters such as the dimensions of a data frame. In this section, we will show you how to get other parameters such as names or sizes. In the following sections, we will discuss how to access rows and columns, as well as individual cells.

Dimensions

Getting the number of rows and columns of a data frame is straightforward:

```
weather numberOfRows. "5"
weather numberOfColumns. "3"
```

You can also get both dimensions of a data frame as a `Point`:

```
weather dimensions. "5@3"
```

Row and column names

Every data frame has names associated with its rows and columns (either used-defined or auto-generated). These names can be used for referencing specific rows or columns (which you will learn in Section 2.18). Let's get the collection of all row and column names of our weather data frame:

```
weather rowNames.
weather columnNames.
```

Transposed DataFrame

Sometimes it is useful to transpose a data frame made out of columns and rows into rows and columns. To do that, you can simply write:

```
weather transposed
```

The result will be a new data frame which looks like this:

	1:10 am	1:30 am	1:50 am	2:10 am	2:30 am
temperature	2.4	0.5	-1.2	-2.3	3.2
precipitation	true	true	true	false	true
type	rain	rain	snow	-	rain

2.18 Accessing rows and columns

In this section, I will show you how to get the values of specific rows and columns, as well as how to modify these values. Rows and columns of a data frame can be accessed either by their names or their numeric indexes.

Accessing by name

You can get row `01:50` or the `temperature` column of the weather data frame by writing:

```
weather row: '01:50' asTime.
weather column: #temperature.
```

Use methods `row: put:` and `column: put:` to modify the values stored in a row or column:

```
weather row: '01:50' asTime put: #(10 true rain).
weather column: #temperature put: #(1.2 -2.1 3.4 -5.9 -0.4).
```

	temperature	precipitation	type
1:10 am	1.2	true	rain
1:30 am	-2.1	true	rain
1:50 am	3.4	true	rain
2:10 am	-5.9	false	-
2:30 am	-0.4	true	rain

If you reference a row or column by a non-existing name you will get the `NotFoundError` and if the array you provide is too big or too small, the `SizeMismatch` error will be signaled. Notice that you can not add a new column using `dataFrame column: #newName put newArray`. This is done with the `addColumn:` set of methods that will be described in the following sections.

Accessing by index

Rows and columns can also be accessed by their numeric indices. You can get the same row and column as in the previous example using:

```
weather rowAt: 3.
weather columnAt: 1.
```

To modify them, use `rowAt: put:` and `columnAt: put:`.

```
weather rowAt: 3 put: #(-1.2 true snow).
weather columnAt: 1 put: #(2.4 0.5 -1.2 -2.3 3.2).
```

	temperature	**precipitation**	**type**
1:10 am	2.4	true	rain
1:30 am	0.5	true	rain
1:50 am	-1.2	true	snow
2:10 am	-2.3	false	-
2:30 am	3.2	true	rain

Accessing multiple rows and columns

To access multiple rows or columns at the same time, you have to provide an array of names or indices, or specify the range of indices. In the following example, we get the three middle rows of the weather data frame:

```
weather rows: { '01:30' asTime . '01:50' asTime . '02:10' asTime}.
weather rowsAt: #(2 3 4).
weather rowsFrom: 2 to: 4.
```

The same can be done to access, for example, the last two columns:

```
weather columns: #(precipitation type).
weather columnsAt: #(2 3).
weather columnsFrom: 2 to: 3.
```

The result will be another data frame with only the requested rows or columns in the order in which you ask for them. This means that you can ask for rows 3, 2, and 5 or the `precipitation` column followed by `temperature`:

```
weather rowsAt: #(3 2 5).
weather columns: #(precipitation temperature).
```

All the above methods can be used together with `put:` to replace the given rows or columns with new ones:

```
DataFrame >> rows: put:
DataFrame >> rowsAt: put:
DataFrame >> rowsFrom: to: put:
DataFrame >> columns: put:
DataFrame >> columnsAt: put:
DataFrame >> columnsFrom: to: put:
```

Head and Tail

To understand the nature of a dataset with 100,000 rows, it helps if we can take a look at its first or last 5 rows. This is called the **head** or **tail** of a dataset. `DataFrame` allows you to get an arbitrary number of rows at its beginning or end using the `head: aNumber` and `tail: aNumber` methods. It also provides the simpler methods `head` and `tail` which return 5 rows.

For example, if we want to get the first 2 rows from our weather data frame:

```
weather head: 2.
```

	temperature	precipitation	type
1:10 am	2.4	true	rain
1:30 am	0.5	true	rain

Or the tail with the last 3 rows:

```
weather tail: 2.
```

	temperature	precipitation	type
1:50 am	-1.2	true	snow
2:10 am	-2.3	false	-
2:30 am	3.2	true	rain

Both `head` and `tail` return new data frames with only the corresponding rows. DataSeries also implements the `head`/`tail` methods, which allows you to get the `head` or `tail` of a single row or column as a new data series with only the first or last elements:

```
(weather column: #temperature) head: 2.
```

	temperature
1:10 am	2.4
1:30 am	0.5

2.19 Adding rows and columns

You can add new rows and columns to a data frame by providing an array of values, a name, and its position as an index:

```
weather
  addColumn: #(86 79 23 16 90)
  named: #humidity
  atPosition: 2.
```

The above statement will modify the weather dataset by adding a humidity column right after the temperature column:

	temperature	humidity	precipitation	type
1:10 am	2.4	86	true	rain
1:30 am	0.5	79	true	rain
1:50 am	-1.2	23	true	snow
2:10 am	-2.3	16	false	-
2:30 am	3.2	90	true	rain

You can omit the position using methods `addRow:named:` or `addColumn:named:`. By default rows and columns will be added to the end of a data frame:

```
weather
  addRow: #(2.0 81 true rain)
  named: '2:50' asTime.
```

	temperature	humidity	precipitation	type
1:10 am	2.4	86	true	rain
1:30 am	0.5	79	true	rain
1:50 am	-1.2	23	true	snow
2:10 am	-2.3	16	false	-
2:30 am	3.2	90	true	rain
2:50 am	2.0	81	true	rain

Alternatively, you can add new rows or columns as data series. In this case you do not need to provide a name because the data series already knows its name.

```
wind := DataSeries
  withValues: #(39 39 32 24 14 14)
  name: #wind.

weather
  addColumn: wind
  atPosition: 2.
```

Notice that we did not specify the keys of the data series with wind measurements. It was initialized with default keys. When adding new row or column as a data series, `DataFrame` does not take keys into account.

	temperature	wind	humidity	precipitation	type
1:10 am	2.4	39	86	true	rain
1:30 am	0.5	39	79	true	rain
1:50 am	-1.2	32	23	true	snow
2:10 am	-2.3	24	16	false	-
2:30 am	3.2	14	90	true	rain

2:50 am	2.0	14	81	true	rain

You can use `addRow:` or `addColumn:` without specifying a position, in which case the new row or column will be added to the end.

Just as before, when we were modifying existing rows and columns, if you try to add a new row or column which is too big or too small, `DataFrame` will signal a `SizeMismatch` error.

2.20 Removing rows and columns

To remove a row or column from a data frame, you have to provide either its name or index. In the previous section, we added two columns and one row to the data frame. Now we can remove the `wind` column and the last row that was added:

```
weather removeColumn: #wind.
weather removeRowAt: 6.
```

This gives us the same weather dataset as we had at the beginning, but with an additional `humidity` column.

	temperature	humidity	precipitation	type
1:10 am	2.4	86	true	rain
1:30 am	0.5	79	true	rain
1:50 am	-1.2	23	true	snow
2:10 am	-2.3	16	false	-
2:30 am	3.2	90	true	rain

If you try to remove a column or row by a name which does not exist or by an index that is out of range, you will get the `NotFoundError` or `SubscriptOutOfBounds`.

2.21 Enumerating rows of a data frame

A `DataFrame` can be treated as an array of rows. You can enumerate them in the same way you would enumerate any other sequenceable collection in Smalltalk: `do:`, `collect:`, `select:`, `reject:`, and `inject:into:`.

do: aBlock

Evaluate `aBlock` for each row of the receiver data frame. Let us convert the temperatures from Celsius to Fahrenheit:

```
weather do: [ :row |
  row at: #temperature transform: [ :celsius |
    celsius * 9/5 + 32 ] ].
```

	temperature	humidity	precipitation	type
1:10 am	36.32	86	true	rain
1:30 am	32.9	79	true	rain
1:50 am	29.84	23	true	snow
2:10 am	27.86	16	false	-
2:30 am	37.76	90	true	rain

select: aBlock

Evaluate `aBlock` for each row of the receiver data frame. Collect into a new data frame only those rows for which `aBlock` evaluates to `true`. Answer the new data frame. We can select the records that are below freezing temperature:

```
weather select: [ :row |
  (row at: #temperature) < 32 ].
```

	temperature	humidity	precipitation	type
1:50 am	29.84	23	true	snow
2:10 am	27.86	16	false	-

reject: aBlock

Evaluate `aBlock` for each row of the receiver data frame. Collect into a new data frame only those rows for which `aBlock` evaluates to `false`. Answer the new data frame. Now, if we reject freezing temperatures, we will get all the other records.

```
weather reject: [ :row |
  (row at: #temperature) < 32 ].
```

	temperature	humidity	precipitation	type
1:10 am	36.32	86	true	rain
1:30 am	32.9	79	true	rain
2:30 am	37.76	90	true	rain

collect: aBlock

Evaluate `aBlock` for each row of the receiver data frame. Collect into a new data frame the rows that were answered by `aBlock`. Therefore, `aBlock` is expected to return a data series of a certain size. This size can be different than the number of columns in the original data frame, but the same for every answered row. Keys of the answered data series will be set as column names of the new data frame. For example, we can collect rows of the weather data frame into a new data frame with the values of the `humidity` column converted to the `0-1` scale and with the `precipitation` column removed:

```
weather collect: [ :row |
  row at: #humidity transform: [ :percent | percent / 100 ].
  row removeAt: #precipitation.
  row ].
```

The result will be a new data frame:

	temperature	humidity	type
1:10 am	36.32	0.86	rain
1:30 am	32.9	0.79	rain
1:50 am	29.84	0.23	snow
2:10 am	27.86	0.16	-
2:30 am	37.76	0.9	rain

Remember that the original weather data frame is not modified.

detect: aBlock

Evaluate `aBlock` for each row of the receiver data frame. Answer the first row for which `aBlock` evaluates to `true`. If none evaluates to `true`, report an error. For example, we can detect the first row with a freezing temperature:

```
weather detect: [ :row |
  (row at: #temperature) < 32 ].
```

	1:50 am
temperature	29.84
humidity	23
precipitation	true
type	snow

detect: aBlock ifNone: exceptionBlock

Evaluate `aBlock` for each row of the receiver data frame. Answer the first row for which `aBlock` evaluates to `true`. If none evaluates to `true`, evaluate `exceptionBlock` which must be a block requiring no arguments. For example, since the weather dataset does not contain any observations (rows) with temperatures lower than 20, the following expression will evaluate to the *not found* string:

```
weather
  detect: [ :row | (row at: #temperature) < 20 ]
  ifNone: [ 'not found' ].
```

inject: thisValue into: binaryBlock

Evaluate `binaryBlock` once for each row in the receiver data frame. The block has two arguments: the second one is the row from the receiver; the first one is the value of the previous evaluation of the block, starting with the argument `thisValue`. Answer the final value of a block.

We can use it to sum up all values of the two numeric columns: `temperature` and `humidity`:

```
(weather columns: #(temperature humidity))
  inject: 0
  into: [ :sum :row | sum + row ].
```

	(no name)
temperature	164.68
humidity	294

Additionally, `DataFrame` provides special methods for enumerating by index. The following table lists all enumeration methods understood by `DataFrame`:

Row	**Row and index**
do:	withIndexDo:
select:	withIndexSelect:
reject:	withIndexReject:
collect:	withIndexCollect:
detect:	-
detect: ifNone:	-
inject: into:	-

We do not need to enumerate rows with their names because each row is a data series which already knows its name.

2.22 Aggregation and Grouping

Aggregation and grouping is among the most used operations in data analysis workflows. It allows us to first group rows of a data frame by a value of some column and then aggregate each one of these groups into a single value with some function or block, for example, `average`, `sum`, etc.

Let's ask the weather data frame to answer the following question: *"What is the average temperature when it rains, snows, or when there is no precipitation?"*. To do that, we group the values of `temperature` column by the `type` column and then find the average value of each group.

```
weather
  group: #temperature
  by: #type
  aggregateUsing: #average
  as: #averageTemperature.
```

This gives us a data series of average temperature by precipitation type called `averageTemperature`:

	averageTemperature
-	27.86
rain	35.66
snow	29.84

You can omit the `as: #averageTemperature` part of that message, in which case the answered data series will have the same name as the column that was aggregated: `temperature`.

Values of data series are grouped into a data series of groups where each group is also a data series. An aggregation function or block is then applied to each one of these groups. Which means that the aggregation function can be any selector understood by `DataSeries`. And the result of the aggregation and grouping expression will be a data series of answers from your aggregation function or block (which can be scalar values or collections).

Let's look at the total number of rows, lowest and highest temperature values, and average humidity in each group:

```
weather
  groupBy: #type
  aggregate: {
    #temperature using: #size as: #count .
    #temperature using: #min as: #minTemperature .
    #temperature using: #max as: #maxTemperature .
    #humidity using: #average as: #avgHumidity }.
```

Notice that the `count` column is constructed by aggregating groups of `temperature` with the `#size` message. In fact, any column can be used in place of `temperature`.

	count	minTemperature	maxTemperature	avgHumidity
-	1	27.86	27.86	16
rain	3	32.9	37.76	85
snow	1	29.84	29.84	23

2.23 Handling nil values

DataFrames are a powerful tool for working with structured data in Pharo. They allow us to organize, manipulate, and analyze data efficiently. However, real-world datasets often contain missing or undefined values, represented as "nil" in Pharo. Handling nil values appropriately is crucial to ensure accurate and reliable data analysis. In this section, we will explore various methods available in Pharo's DataFrame package for handling nil values effectively.

Identifying nil values

Before we can handle nil values, it is essential to identify their presence within a data frame. Pharo's DataFrame package provides these methods for detecting nil values:

- `hasNils`: This method returns true if there is at least one nil value in the data frame.
- `hasNilsByColumn` : Returns a dictionary indicating the presence of any nil values column-wise. The keys of the dictionary represent the column names, and the values (true or false) indicate whether nil values exist in the corresponding column.
- `numberOfNils` : Returns a dictionary indicating the number of nil values column-wise. The keys of the dictionary represent the column names, and the values represent the count of nil values in each column.

Suppose we had this data frame :

```
weather := DataFrame withRows: #(
  (2.4 true rain)
  (0.5 true nil)
  (-1.2 true snow)
  (-2.3 nil nil)
  (3.2 true rain)).

weather columnNames: #(temperature precipitation type).

weather rowNames: #( '01:10' '01:30' '01:50' '02:10' '02:30').
```

Since this is a small data frame, it can easily be seen that it has nil values, however if the data frame is large, it will be difficult to physically check the data frame for nil values. This is where `hasNils` becomes useful :

```
weather hasNils. "true"
```

You can also see in which columns these nil values are present :

```
weather hasNilsByColumn.
```

key	value
precipitation	true
type	true
temperature	false

You can even find out the number of nil values in each column :

```
weather numberOfNils.
```

key	value
precipitation	1
type	2
temperature	0

Removing nil values

When dealing with nil values, it may be necessary to remove or filter out rows or columns containing these values. It should be noted that usually in Data Science and Machine Learning tasks, columns are removed only if there are many nil values in that column or if the column doesn't contain a lot of information that helps your analysis and rows are removed if the number of rows with nil values is very less compared to the total number of rows. The following methods assist in removing nil values:

- `removeColumnsWithNilsAtRow` : Removes all columns with nil values at a specified row number from the data frame.
- `removeColumnsWithNilsAtRowNamed` : Removes all columns with nil values at a specified row name from the dataframe.
- `removeRowsWithNils` : Removes all rows from the data frame that have at least one nil value.
- `removeRowsWithNilsAtColumn` : Removes all rows with nil values at a specified column number from the data frame.
- `removeRowsWithNilsAtColumnNamed` : Removes all rows with nil values at a specified column name from the data frame.

If you want to remove all columns which have their second value as a nil value :

```
weather removeColumnsWithNilsAtRow: 2.
```

	temperature	precipitation
1:10 am	2.4	true
1:30 am	0.5	true
1:50 am	-1.2	true
2:10 am	-2.3	nil
2:30 am	3.2	true

You can remove all rows from a data frame which have at least one nil value :

```
weather removeRowsWithNils.
```

	temperature	precipitation	type
1:10 am	2.4	true	rain
1:50 am	-1.2	true	snow
2:30 am	3.2	true	rain

If you want to remove rows which have nil values in the column 'precipitation' :

```
weather removeRowsWithNilsAtColumnNamed: 'precipitation'.
```

	temperature	precipitation	type
1:10 am	2.4	true	rain
1:30 am	0.5	true	nil
1:50 am	-1.2	true	snow
2:30 am	3.2	true	rain

Replacing nil values

In certain cases, it might be more appropriate to replace nil values with meaningful alternatives. It is important to note that the choice of replacement method depends on the nature of the data and the specific analysis goals. Different replacement strategies can be applied based on the characteristics and patterns of the missing data. Pharo's DataFrame package provides various methods for replacing nil values with the user's desired alternatives as well as statistical alternatives :

- `replaceNilsWith: anObject` : Replaces all nil values in the data frame with the provided object, `anObject`.
- `replaceNilsWithAverage` : Replaces all nil values in the data frame with the average value of the column in which they are present.
- `replaceNilsWithMedian` : Replaces all nil values in the data frame with the median of the column in which they are present.
- `replaceNilsWithMode` : Replaces all nil values in the data frame with the mode of the column in which they are present.
- `replaceNilsWithNextRowValue` : Replaces all nil values in the data frame with the value of the next non-nil element in the same column. If the last value of a column in the data frame is nil, it will remain nil even after using this method because there is no value after it in the column which can replace it.
- `replaceNilsWithPreviousRowValue` : Replaces all nil values in the data frame with the value of the previous non-nil element in the same column. If the first value of a column in the data frame is nil, it will remain nil even after using this method because there is no value before it in the column which can replace it.

- `replaceNilsWithZero` : Replaces all nil values in the data frame with zero.

If you want to replace all nil values in a data frame with 'value' :

```
weather replaceNilsWith: 'value' .
```

	temperature	**precipitation**	**type**
'01:10'	2.4	true	rain
'01:30'	0.5	true	value
'01:50'	-1.2	true	snow
'02:10'	-2.3	value	value
'02:30'	3.2	true	rain

You can replace all nil values in a column with a statistical value of that column, this example replaces nil values with the mode value of that column :

```
weather replaceNilsWithMode .
```

	temperature	**precipitation**	**type**
'01:10'	2.4	true	rain
'01:30'	0.5	true	rain
'01:50'	-1.2	true	snow
'02:10'	-2.3	true	rain
'02:30'	3.2	true	rain

If you want to replace nil values with the previous non-nil value in the same column :

```
weather replaceNilsWithNextRowValue .
```

	temperature	**precipitation**	**type**
'01:10'	2.4	true	rain
'01:30'	0.5	true	snow
'01:50'	-1.2	true	snow
'02:10'	-2.3	true	rain
'02:30'	3.2	true	rain

2.24 Reading from and writing to files

In practice, data frames are useful when you can load some data from an external source into a data frame, modify and analyse it, and then save the result. The external source can be a CSV or Excel file, a database connection, or any other source of data. In this section, I will describe the `DataFrame-IO` package of the `DataFrame project` which allows you to load and save data frames.

DataFrame-IO provides two abstract classes: DataFrameReader and DataFrameWriter. Each of them has only one abstract method: DataFrameReader >> readFrom: aLocation (expected to return a data frame) and DataFrameWriter >> write: aDataFrame to: aLocation. To add support for some external data source, you need to override those methods in your subclasses, providing the actual algorithm for reading and writing data. DataFrame has two methods that allow you to read or write it using your reader or writer:

```
DataFrame class >> readFrom: aLocation using: aDataFrameReader
  "Read data frame from a given location using a given
    DataFrameReader. Location can be a file reference, a database
    connection, or something else (depending on the implementation
    of the reader)"
  ^ aDataFrameReader readFrom: aLocation
```

```
DataFrame >> writeTo: aLocation using: aDataFrameWriter
  "Write data frame to a given location using a given
    DataFrameWriter. Location can be a file reference, a database
    connection, or something else (depending on the implementation
    of the writer)"
  aDataFrameWriter write: self to: aLocation
```

CSV support

CSV (comma-separated values) is the file format that is most commonly used by data scientists to save and share tabular data. It is a simple text file in which each row is written on a new line and values of a row are separated by commas. You can use another character as the separator instead of comma; for example, one common choice is tab.

The DataFrame-IO package comes with two subclasses of DataFrameReader and DataFrameWriter: DataFrameCsvReader and DataFrameCsvWriter. They override readFrom: and write:to: methods using NeoCSV and allow you to specify optional configurations, such as the separator character (comma by default) or line end convention (either #cr, #lf, or #crlf - defaults to OS convention). By default, DataFrameCsvReader will read all columns of the CSV file as data frame columns and assign the default names to rows - numbers from 1 to numberOfRows. However, you can configure it to read the first column as row names by sending true to the DataFrameCsvReader >> includeRowNames:. In this case aLocation parameter of readFrom: and write:to: methods is expected to be a FileReference.

Because reading and writing CSV files is very common in a data analysis workflow, DataFrame provides shortcuts for these methods:

```
DataFrame class >> readFromCsv:
DataFrame class >> readFromCsv: withSeparator:
DataFrame class >> readFromCsvWithRowNames:
DataFrame class >> readFromCsvWithRowNames: separator:
```

```
DataFrame >> writeToCsv:
DataFrame >> writeToCsv: withSeparator:
```

2.25 Conclusion

This chapter has guided you through the complete functionality of the `DataFrame` project. It offered you examples of different methods without any logical connection between them. In the following chapters you will find short tutorials which demonstrate the application of data frames to real life problems.