The dataChest For Dummies

Alexander Opremcak*

April 21, 2016

Contents

*electronic address: opremcak@wisc.edu

1 Introduction

The dataChest is a Python library written for storing and retrieving datasets. It enforces good user habits while remaining flexible enough to deal with data of virtually any type or shape. In this document, I will briefly describe all of the methods this class has to offer. I will then formalize our definitions of 1D and 2D datasets while providing examples of how to enter each of the types that we discuss.

2 Overview of functionality

The dataChest has a small number of public methods which can be broken up into three categories: i) Folder Creation and Navigation methods, ii) Dataset methods, and iii) Parameter methods. The Folder Creation and Navigation methods give users basic tools for navigating and creating directories. The Dataset methods allow users to create new datasets, open existing datasets, and fetch stored data. Parameter methods allow users to attach additional information, 'parameters', to a dataset.

Before we begin with any examples of how to use these methods, we must set up two environment variables: `DATA_CHEST_ROOT` and `PYTHONPATH`. The `DATA_CHEST_ROOT` will provide the dataChest with the starting location (root directory) in which all folders and datasets created with the dataChest will be placed. Make sure that this location exists before you attempt to run the Python code examples. The `PYTHONPATH` enviroment variable points to the location of the servers repository where the dataChest is located. By creating this environment variable, you will be able to import dataChest from any directory. If you are working on a mac, open up terminal and type:

```
cd $HOME
ls -a
```

This will list all of the contents within your home directory, including hidden files. You should see a file called `.bash_profile` (or `.profile` for older macs). Now type

```
nano .bash_profile
```

which will open a text editor for modifying this file. Once opened enter the following two lines:

```
DATA_CHEST_ROOT=$HOME/Desktop/Data
export DATA_CHEST_ROOT
PYTHONPATH=${PYTHONPATH}:$HOME/Desktop/Repositories/servers/dataChest
export PYTHONPATH
```

After you save these changes, run

```
source .bash_profile
```

to make your changes take effect. Now type

```
python
```

into terminal and instantiate a dataChest object as follows:

```
from dataChest import *
d = dataChest("DataChestTutorial") #sets root for this instance
```

The folder "DataChestTutorial" need not already exist, it is created automatically if non-existent. From here on I will assume that the **dataChest** has been imported, the object **d** above has been instantiated, and each snippet of code has been executed in the order presented.

2.1 Folder creation and navigation methods

2.1.1 mkdir()

mkdir() allows users to make new directories. Note that **mkdir**() does not automatically change your working directory when a new directory is created nor would it in Linux/Unix. Let us see how it works:

```
d.mkdir("DummyDirectory") #creates directory if non-existent
```

If the directory already exists, the method simply raises an exception.

2.1.2 ls()

ls() lists the contents, files and folders, of the working directory. Here is how its used:

```
contents = d.ls() # contents = [files, folders]
files = contents[0] # list of all files
folders = contents[1] # list of all folders

print "DummyDirectory" in folders # ==> True
print "NonexistentDirectory" in folders # ==> False

print "contents=", contents # ==> [[],["DummyDirectory"]]
print "files=", files # ==> [] no files created thusfar
print "folders=", folders # ==> ["DummyDirectory"]
```

2.1.3 pwd()

pwd() returns the path of the working directory. This method will be used in the example that follows.

2.1.4 cd()

cd() is used to change the working directory. Try running this:

```
print "working directory =", d.pwd() # prints working directory
d.cd("DummyDirectory") # move to folder just created
print "working directory =", d.pwd()
d.mkdir("DummySubDirectory") # created within "Dummy Directory"
d.cd("..") # bumps us up 1 directory placing us in root
print "working directory =", d.pwd()
d.cd(["DummyDirectory", "DummySubDirectory"]) # list style input
print "working directory =", d.pwd()
d.cd("") # short-cut for moving us up to root directory
print "working directory =", d.pwd() # root
```

Note that the root directory is `DATA_CHEST_ROOT` + "/" + "DataChestTutorial", not simply `DATA_CHEST_ROOT`. This was done to avoid cluttering the root directory in which all of our datasets will be stored. Also note that **cd**() accepts string and list style inputs.

2.2 Dataset methods

2.2.1 createDataset()

createDataset() is used to create datasets. It requires users to enter a name for the dataset, independent variable information, and dependent variable information. Here is a quick example of how it is used:

```
d.createDataset("voltageTimeSeries",
                [("ind1", [1], "float64", "s")],
                [("dep1", [1], "float64", "V")]
                )
```

Lets have a look at the arguments. The first argument "voltageTimeSeries" is the name of the dataset. This string must contain valid filename characters only. The second argument [("ind1", [1], "float64", "S")] is a list of independent variable information. If you had multiple independent variables, this list would have multiple tuple entries, one tuple per independent variable. Each tuple contains variable information, in particular a name (`"ind1"`), shape (`[1]`), type (`"float64"`), and units (`"s"`). The third argument is the analog of this but for dependent variables.

While scalar data is very common, it pays to allow for other shapes of data to be stored. For example, suppose you were interested in looking at the dependence of some transmission measurement as a function of gate bias on some device. So you fix a gate bias, take a transmission measurement, then you fix a new gate bias, take a transmission measurement and so on. At the end of the day you have a dataset that is a collection of scalars (the gate biases) and arrays (the transmission waveforms). This type of data could be saved by creating the following dataset:

```
d.createDataset("newDataShape",
                [("GateBias", [1], "float64", "V"),
                 ("Frequency", [1000], "float64", "Hz")],
                [("Transmission", [1000], "complex128", "dB")]
                )
```

where we now have a shape of `[1000]`. This really generalizes our notion of 'variables' to something that can take on arbitrary shape. Shapes are then just lists which specify the dimensionality of an N-dimensional array. For example 3x3 matrix would have shape `[3,3]` where as a 1000 point waveform (1D array) has shape `[1000]`.

The data type specifies the type of data that makes up the elements of these N-dimensional arrays. Acceptable data types are given in the following table:

Table 1: Data Types

Data Types	Description#1
`"bool_"`	Boolean
`"int8"`	8-bit Signed Integer (-2^7 to 2^7-1)
`"int16"`	16-bit Signed Integer (-2^{15} to $2^{15}-1$)
`"int32"`	32-bit Signed Integer (-2^{31} to $2^{31}-1$)
`"int64"`	64-bit Signed Integer (-2^{63} to $2^{63}-1$)
`"uint8"`	8-bit Unsigned integer (0 to 2^8-1)
`"uint16"`	16-bit Unsigned integer (0 to $2^{16}-1$)
`"uint32"`	32-bit Unsigned integer (0 to $2^{32}-1$)
`"uint64"`	64-bit Unsigned integer (0 to $2^{64}-1$)
`"int_"`	Default integer type (normally either `"int64"` or `"int32"`)
`"float16"`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`"float32"`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`"float64"`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`"float_"`	Shorthand for `"float64"`
`"complex64"`	Complex number, represented by two 32-bit floats
`"complex128"`	Complex number, represented by two 64-bit floats
`"complex_"`	Shorthand for `"complex128"`
`"utc_datetime"`	UTC Datetime Float
`"string"`	String

The data type `"utc_datetime"` deserves some special attention. UTC, which stands for Coordinated Universal Time, is the primary time standard by which the world regulates clocks and time. Although this may seem like overkill, it was chosen to eliminate any and all ambiguity that may be introduced when it comes to storing datetimes in your datasets. Here is how you generate the current `"utc_datetime"`:

```
from dateStamp import * # helper class for dataChest
dStamp = dateStamp() # instantiation
```

```
utcFloat = dStamp.utcNowFloat()
print "utcFloat=", utcFloat # ==> current UTC time as a float
```

Note this has microsecond timing resolution.

2.2.2 getDatasetName()

getDatasetName() returns the name of the current dataset.

```
datasetName = d.getDatasetName()
print "datasetName=", datasetName #dateStamp+"_"+"newDataShape.hdf5"
```

Note that a dateStamp is appended to the front of the name you provided **createDataset**() with on your most recent call. This dateStamp plus the name provided to **createDataset**() make up the actual name of the dataset on disk. This dateStamp makes the filename unique while preserving the chronological order in which the dataset was created with respect to others in the same folder.

2.2.3 getVariables()

getVariables() returns a list of independent and dependent variables lists as they were given when the dataset was created. Try the following:

```
varsList = d.getVariables()
indepVarsList = varsList[0]
depVarsList = varsList[1]

print indepVarsList # list of independent vars
# ==> [("GateBias", [1], "float64", "s"),
# ("Frequency", [1000], "float64", "Hz")]
print depVarsList # list of dependent vars
# ==> [("Transmission", [1000], "complex128", "dB")]
```

2.2.4 addData()

addData() is used for adding data to a newly[1] created dataset. Run the following:

```
d.createDataset("someDataset",
                [("ind1", [1], "float64", "s")],
                [("dep1", [1], "complex128", "V")]
                )
d.addData([[1.0, 2.0+1j*2.0]]) #add 1 row
d.addData([[2.0, 4.0+1j*4.0],[3.0, 6.0+1j*6.0]]) #add 2 rows
```

It is worth noting that only one dataset can be opened per dataChest object, meaning that if you wanted to create and add to multiple datasets at once, each dataset would require its own dataChest object. Here is an example of how to do precisely this:

[1] Adding data to an old dataset will be discussed with the **openDataset**() method.

```
d1 = dataChest("ParallelWrites")
d2 = dataChest("ParallelWrites")

d1.createDataset("Dataset1",
                 [("LoggerTime", [1], "utc_datetime", "")],
                 [("ColdStageTemp", [1], "float64", "K")]
                 )

d2.createDataset("Dataset2",
                 [("time", [1], "float64", "s")],
                 [("voltage", [1], "float64", "V")]
                 )

d1.addData([[dStamp.utcNowFloat(), 21.2e-3]]) # write to one
d2.addData([[1.0, 2.0]]) # then to the other

d1.addData([[dStamp.utcNowFloat(), 19.4e-3]])  # repeat
d2.addData([[2.0, 4.0]])
```

2.2.5 getData()

getData() is used for retrieving data from an open dataset. Whether this dataset was newly created or recently re-opened for inspection is immaterial. Assuming we carried out the example above let's run the following code

```
data = d.getData() #fetch all data from "someDataset"
data # all entries
data[0] # ==> [ 1.+0.j  2.+2.j]
data[1] # ==> [ 2.+0.j  4.+4.j]
data[2] # ==> [ 3.+0.j  6.+6.j]
```

There may be situations in which a user wishes to fetch a subset of the available data as opposed to the all or nothing style shown above. The **getData**() method permits such behavior in a manner similar to how arrays are sliced. Here are some examples:

```
d.getData(0) # first element
d.getData(1) # second element
d.getData(-1) # last element
d.getData(-2) # second to last element
d.getData(None, 2) # equivalent to [:2]
d.getData(1, None) # equivalent to [1:]
```

Basically, `None` as the first (last) argument plays the role of blank followed by (preceded by) a colon in the usual list slicing syntax.

2.2.6 openDataset()

openDataset() is used for opening existing, i.e. datasets that were already created and are no longer the current dataset with respect to the dataChest object of interest, take **d** for example. Let's run the following:

```
currentDataset = d.getDatasetName() # get current dataset name
print currentDataset
d.createDataset("someName",  # this is now the current dataset
                [("ind1", [1], "float64", "Seconds")],
                [("dep1", [1], "float64", "Volts")]
                ) #create new dataset
print d.getDatasetName() # name differs from currentDataset
d.openDataset(currentDataset) # reopens the previous dataset
print d.getDatasetName() # currentDataset
d.addData([[4.0, 8.0+1j*8.0]]) #raises exception
print d.getData() # prints data from old dataset
```

openDataset() does not allow users to use **addData**() by default. This was done purposefully so that datasets are not unintentionally written to when a user is being sloppy in software. If a user would like to add data[2] to an existing dataset, they must call **openDataset**() with an optional parameter specifying that they would like modification privileges as follows:

```
currentDataset = d.getDatasetName() #get current dataset name
d.createDataset("someName", # this is now current dataset
                [("ind1", [1], "float64", "Seconds")],
                [("dep1", [1], "float64", "Volts")]
                ) #create new dataset
d.openDataset(currentDataset, modify = True) #write/read access
d.addData([[4.0, 8.0+1j*8.0]]) #add to an old dataset with modify = True
```

2.3 Parameter methods

2.3.1 addParameter()

addParameter() is used for appending parameters to the current dataset. Here is how it works:

```
d.addParameter("Who", "Mike Jones")
d.addParameter("Time", "Goon Time")
d.addParameter("Base Temp", -13.789) #et cetera
```

If a user attempts to add a parameter that already exists, this method will throw an exception. If a user would like to overwrite an existing parameter with a new value, even new types are allowed, then they must opt for overwrite privileges using the following syntax:

[2]Parameters can also be added to datasets opened with **openDataset**() using the same prescription.

```
d.addParameter("DummyParam", 11)
d.addParameter("DummyParam", 14)  #raises an exception
d.addParameter("DummyParam", 13, overwrite = True)
d.addParameter("DummyParam", "blah", overwrite = True)
```

2.3.2 getParameter()

getParameter() is used for retrieving the value of an existing parameter. Here is how it works:

```
print d.getParameter("Who") # ==> "Mike Jones"
print d.getParameter("Time") # ==> "Goon Time"
print d.getParameter("Base Temp") # ==> -13.789
print d.getParameter("DummyParam") # ==> "blah"
```

2.3.3 getParameterList()

getParameterList() is used for retrieving a list of all available parameters. Here is how it works:

```
print d.getParameterList()
```

3 Canonical datasets

While the overview of user-end functionality may have been helpful, it will prove useful to go over some examples of how to create and add to some of the most commonly encountered types of datasets. For each type of data, I will provide motivation for the definition and then an example of how to add it to the dataChest.

3.1 1D data

By 1D data, we mean a dataset with one independent variable and m dependent variables where $m \geq 1$. Loosely speaking, this captures what it means to be a 1D dataset entirely. In what follows, I will motivate the two most commonly encountered types of 1D data.

3.1.1 Arbitrary 1D data

The most general type of 1D data has an arbitrary spacing between consecutive data points along the independent variable axis, the x-axis. By fixing the independent variable at some value, call it v, and measuring all of the m dependent quantities $Q_1(v), \ldots, Q_m(v)$ of interest, we obtain a single row of data

$$\text{row} = \Big[v, Q_1(v), \ldots, Q_m(v)\Big] \tag{1}$$

Note the size brackets that we use here to define a row. For each value of v, we repeat this process and eventually our data looks like

$$\text{data} = \Big[\Big[v_0, Q_1(v_0), \ldots, Q_m(v_0)\Big], \Big[v_1, Q_1(v_1), \ldots, Q_m(v_1)\Big], \ldots \Big[v_N, Q_1(v_N), \ldots, Q_m(v_N)\Big]\Big] \tag{2}$$

where we have $N+1$ rows of data with $m+1$ columns of scalar[3] data in each row. Let us call data of this format **Arbitrary Type 1 Data**. Here is an example of how this data entered:

```
import numpy as np
d.createDataset("MyFavoriteTimeSeries",
                [("indepName1", [1], "float64", "Seconds")],
                [("depName1", [1], "float64", "Volts")]
                )
d.addParameter("X Label", "Time")
d.addParameter("Y Label", "Digitizer Noise")
d.addParameter("Plot Title", "Random Number Generator")
net = []
for ii in range(0, 100):
    net.append([float(ii),np.random.rand()])
d.addData(net) #add 100 rows of data at once
d.getData() #single row
```

Note that the shape of each variable is `[1]` which is really the definition of **Arbitrary Type 1 Data** as far as the dataChest is concerned.

Alternatively we could group this data like so

$$\text{data} = \Big[\Big[[v_0, \ldots, v_N], [Q_1(v_0), \ldots, Q_1(v_N)], \ldots [Q_m(v_0), \ldots, Q_m(v_N)]\Big]\Big] \tag{3}$$

Note that each column ('variable') has the same length and the implicit functional dependence of the Q_i's as a function of index that we have assumed. Let us call data of this format **Arbitrary Type 2 Data**. Here is how it is entered:

```
res = 1e-4
timeAxis = np.arange(0.0, 1.0, res)
```

[3]String data is also allowed.

```
d.createDataset("DampedOscillations",
                [("time", [len(timeAxis)], "float64", "Seconds")],
                [("Oscillation", [len(timeAxis)], "float64", "Volts")]
                )
d.addParameter("X Label", "Time")
d.addParameter("Y Label", "Voltage")
d.addParameter("Plot Title", "Damped Oscillations")
d.addData([ [timeAxis, np.sin(2 * np.pi * timeAxis)] ])
d.getData()
```

We use the term 'arbitrary' for these types of data because it is not necessary for the v_i's to be related by any closed form expression as of function of array index. So there you have it. All 1D data fall under this category. So were done ... not quite!

3.1.2 1D scans

Suppose we rip a time series from an ADC with some fixed sampling rate, say 1GS/s. Then our independent axis is specified uniquely by 3 numbers, namely the initial time t_0, the final time t_f, and the number of samples N. The time at the j^{th} point in our time series is given by

$$t[j] = t_0 + \left(\frac{t_f - t_0}{N-1}\right) \cdot j \tag{4}$$

Let us call this style of data a **Linear 1D Scan** along the x-axis. In light of this equation, let us store our data in slightly more efficient manner

$$\begin{aligned} \text{data} = \Big[\Big[& [t_0, t_f], [V_0(t_0), V_0(t_1), \ldots, V_0(t_f)], \\ & [V_1(t_0), V_1(t_1), \ldots, V_1(t_f)], \ldots \\ & [V_m(t_0), V_m(t_1), \ldots, V_m(t_f)]\Big]\Big] \end{aligned} \tag{5}$$

where $[t_0, t_f]$ is simply shorthand an N dimensional array whose values are determined by equation (4). Here is how its entered:

```
length =1e7
mu, sigma = 1, 0.1
gaussian = mu + sigma*np.random.randn(length)
t0 = 0.0
tf = 100.0
d.createDataset("LinearWaveform",
                [("indepName1", [2], "float64", "Seconds")],
                [("depName1", [int(length)], "float64", "Volts"),
                 ("depName2", [int(length)], "float64", "Volts")])
```

```
shorthandTime = [t0, tf]
d.addParameter("Scan Type", "Lin")
d.addData([[shorthandTime,gaussian,gaussian]])
d.getData()
```

As another example, suppose we are doing reflection and transmission measurements with a 2-port VNA from 20 MHz to 20 GHz. A linear frequency sweep will place a very low point density in the "low" frequency region with a majority of the points falling in the GHz region of frequency space. One way to get around this is by linearizing your frequency data on a logarithimic scale, i.e. a log sweep. Again, the independent axis, frequency, is determined by 3 numbers, that is the starting frequency f_0, the stopping frequency f_f, and the number of points swept in frequency space N. Then the frequency at the m^{th} point is given by

$$\log(f[m]) = \log(f_0) + \left(\frac{\log(f_f) - \log(f_0)}{N-1}\right) \cdot m \tag{6}$$

Lets call this style of data a **Logarithmic 1D Scan** along the x-axis. Following the above paragraph our data will look like

$$\begin{aligned} \text{data} = \Bigg[\Bigg[& [f_0, f_f], [S_{11}(f_0), S_{11}(f_1), \ldots, S_{11}(f_f)], \\ & [S_{12}(f_0), S_{12}(f_1), \ldots, S_{12}(f_f)], \\ & [S_{21}(f_0), S_{21}(f_1), \ldots, S_{21}(f_f)], \\ & [S_{22}(f_0), S_{22}(f_1), \ldots, S_{22}(f_f)]\Bigg]\Bigg] \end{aligned} \tag{7}$$

where $[f_0, f_f]$ is simply shorthand an N dimensional array whose values are determined by equation (6).

Since 1D scans are really just specific cases of arbitrary 1D datasets, you may wonder why this discussion is justified. However if the distinction being made here seems artificial, consider the fact that data returned from most ADCs, VNAs, and Spectrum Analyzers contains only the y values, the x-axis is built in software and returned for your convenience.

3.2 2D data

By 2D dataset, we mean a dataset with 2 independent variables and m dependent variables. The example that jumps to mind is 2D bias sweep over some rectangular grid but there are many others.

3.2.1 Arbitrary 2D data

Following the section on Arbitrary 1D Data, the most general type of 2D data has and arbitrary spacing between consecutive points with respect to each of

the dependent variables. Meaning you could move along a spiral, raster over some arbitrary 2D shape, or sweep over a rectangle. Suppose we can vary two independent quantities, v and p. We are interested in studying how the m dependent quantities $Q_1(v,p),\ldots,Q_m(v,p)$ vary with v and p. For each combination $(v,p) \in \{(v_0,p_0),(v_1,p_1),\ldots,(v_N,p_N)\}$, we measure each of the Q_i's. The data will look something like

$$\begin{aligned}\text{data} = \Bigg[\Big[&v_0, p_0, Q_1(v_0,p_0),\ldots,Q_m(v_0,p_0)\Big],\\ &\Big[v_1, p_1, Q_1(v_1,p_1),\ldots,Q_m(v_1,p_1)\Big],\ldots\\ &\Big[v_n, p_n, Q_1(v_n,p_n),\ldots,Q_m(v_n,p_n)\Big]\Bigg]\end{aligned} \tag{8}$$

Again lets call data of this format **Arbitrary Type 1 Data** where we now have two independent quantities, v and p, instead of one.

Alternatively we could format it as

$$\begin{aligned}\text{data} = \Bigg[\Big[&[v_0, v_1,\ldots,v_n],[p_0,p_1,\ldots,p_n],\\ &[Q_1(v_0,p_0),Q_1(v_1,p_1),\ldots,Q_1(v_n,p_n)],\\ &[Q_2(v_0,p_0),Q_2(v_1,p_1),\ldots,Q_2(v_n,p_n)],\ldots\\ &[Q_m(v_0,p_0),Q_m(v_1,p_1),\ldots,Q_m(v_n,p_n)]\Big]\Bigg]\end{aligned} \tag{9}$$

Again we call data of this format **Arbitrary Type 2 Data**.

3.2.2 2D scans

While sweeping around in 2D space in an arbitrary manner is fun, it sometimes makes sense to sweep in a more systematic way. Suppose we are interested in rastering over some rectangular grid, namely all pairs of points (v,p) such that $v \in \{v_0, v_1,\ldots,v_m\}$ and $p \in \{p_0,p_1,\ldots,p_n\}$ where each of these sets follows a formula similar to equation (4) or (6). By measuring each of the Q_i's for all pairs, we map out a 2D surface over some rectangular domain that we decide upon in software. Lets fix a point along the v-axis and perform a 1D Scan along the p-axis, then move to the next point along the v-axis and repeat, we obtain

data that looks like

$$\begin{aligned}\text{data} = \Bigg[&\Big[v_0, [p_0, p_n], [Q_1(v_0, p_0), Q_1(v_0, p_1), \ldots, Q_1(v_0, p_n)] \Big] \\ &\Big[v_1, [p_0, p_n], [Q_1(v_1, p_0), Q_1(v_1, p_1), \ldots, Q_1(v_1, p_n)] \Big], \ldots \\ &\Big[v_m, [p_0, p_n], [Q_1(v_m, p_0), Q_1(v_m, p_1), \ldots, Q_1(v_m, p_n)] \Big] \Bigg]\end{aligned} \tag{10}$$

where $[p_0, p_n]$ is compact notation for a linear or logarithmic scan. Let us call this a **2D Scan** with scan direction along the y-axis.

Alternatively, we could have fixed a point along the p-axis and performed a 1D scan along the v-axis, then stepped to the next point on the p axis and so on which would give the data

$$\begin{aligned}\text{data} = \Bigg[&\Big[[v_0, v_m], p_0, [Q_1(v_0, p_0), Q_1(v_1, p_0), \ldots, Q_1(v_m, p_0)] \Big] \\ &\Big[[v_0, v_m], p_1, [Q_1(v_0, p_1), Q_1(v_1, p_1), \ldots, Q_1(v_m, p_1)] \Big], \ldots \\ &\Big[[v_0, v_m], p_n, [Q_1(v_0, p_n), Q_1(v_1, p_n), \ldots, Q_1(v_m, p_n)] \Big] \Bigg]\end{aligned} \tag{11}$$

Again we call this a **2D Scan** with scan direction along the x-axis.

These two forms encapsulate how nearly all buffered 2D scans are actually carried out with instruments. Fix one independent variable, sweep the other while measuring some quantity, step and repeat.