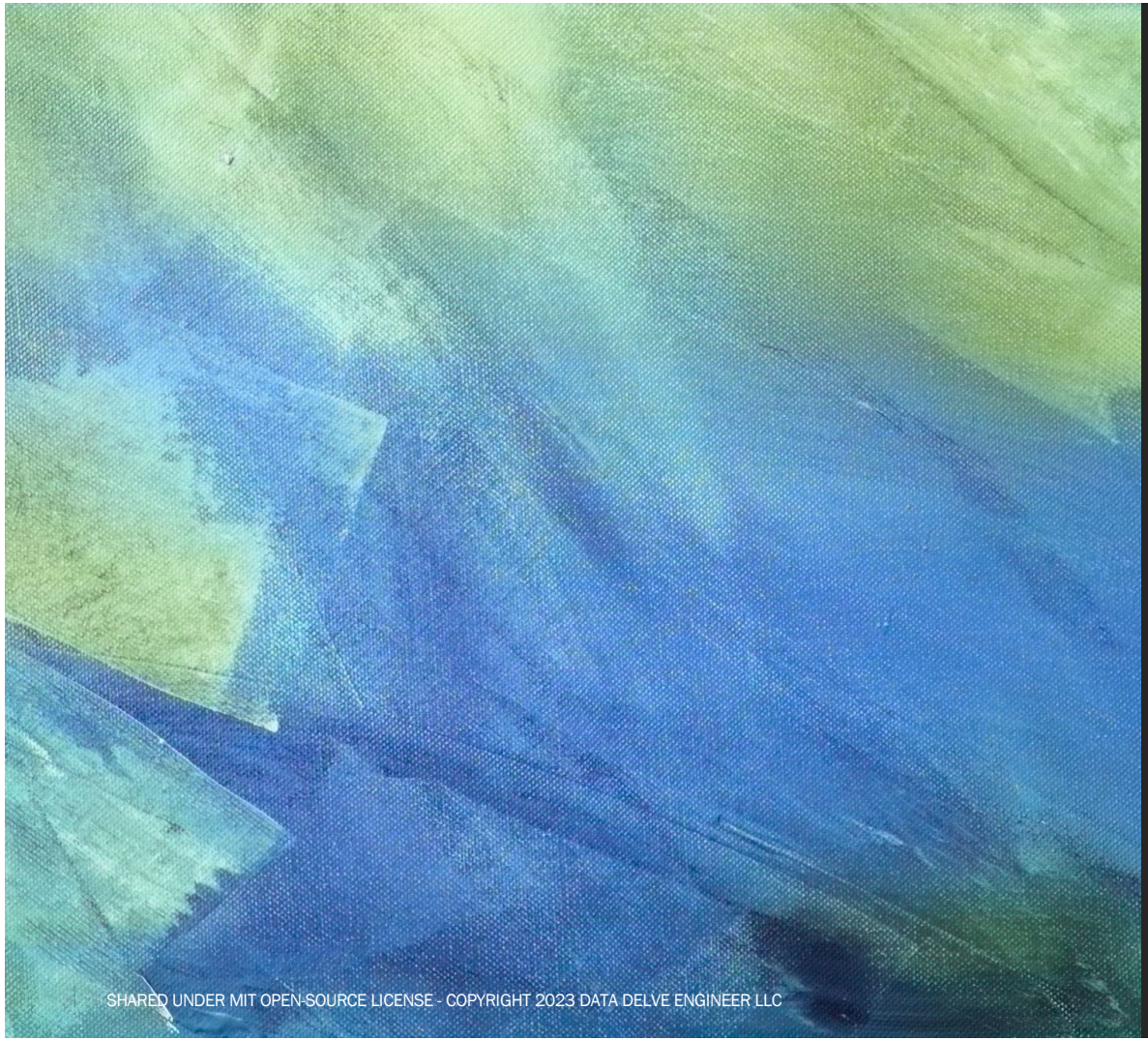


PRACTICAL PYTHON FOR MODELING

J.D. LANDGREBE

DATA DELVE LLC

version July 12, 2023

WORKSHOP LEADER BACKGROUND: J.D. LANDGREBE

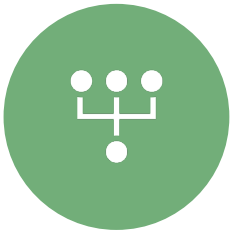
- 34 years as Procter and Gamble R&D technologist
  - Material supply, consumer understanding, IP, R&D
  - Global Modeling Community
  - Training and mentoring
- Chemical Engineering BS and MS
  - Aerosol Science Research and Publications – Particle Size Growth Modeling
- Data science and technical modeling
  - Excel/VBA, JMP and Python software
  - Visualization, and design of experiments
  - PowerBI/DAX
- Consulting Practice: datadelveengineer.com
- LinkedIn www.linkedin.com/in/jdlandgrebe

Data Delve

MOTIVATIONS/INSIGHTS– DO THESE RESONATE?

Good collaboration on models depends on contributors having space within which to work and confidence in how to connect their work with others'

The right amount of code planning cuts down on rework and allows you to work with more complex topics



Picking a model back up from someone else (or from the one-year-ago you) can be challenging

Most models don't have an inspectable validation trail that that builds confidence in them as you work with them and seek to extend them

CHARACTERISTICS OF GOOD MODELS AND ANALYSIS PIPELINES

- Good models and analyses are **curated**. Another knowledgeable human can look at the files and not only repeat the analysis but extend it without the author being present.
- Good models and analyses are **validated**. They come with re-runnable test cases consisting of mocked up data proving correctness of calculations, data cleaning steps and output generation.
- Good models are **maintainable** and **extensible**. They are written in a modular way, so that they are easy to build on and easy to reapply.

TAKEAWAYS FOR YOU FROM THESE WORKSHOPS

- Grow in your ability to architect your projects for personal efficiency and collaboration
  - Learn to distinguish Python code by a 3-level architecture quality scale
  - Practice creating Level 2 (aka Jupyter notebook arranging functions into ”procedures”) code for models
  - Learn basic elements of a Level 3 approach
    - Jupyter “dashboard” + *.py code structure
    - ”Tuhdoop” combination of pre-written Pytest validation (TDD) and object-oriented programming with Python classes (OOP)
- Learn folder structures and projfiles.py file management library helpful for curating and collaboration
  - Model connects naturally to needed data and *.py libraries
  - Toggle freely between test and “production” modes to encourage validation
  - Version control and ability to “use” a model while continuing to develop
- Gain a planning template you can use to get organized

NOMENCLATURE - PRACTICAL DEFINITIONS

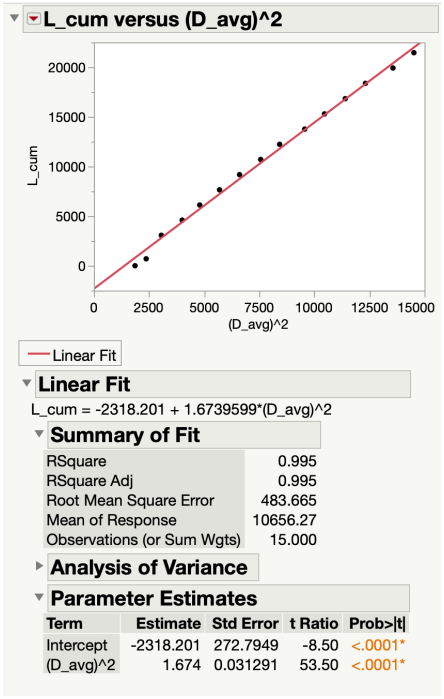
- Curating code and projects
  - Others and the future you can instantly know how a model works and how to dig deeper to get more info
  - The “new hire” test
  - Benchmark: no meeting needed with model owner/developer to get going with using and extending a model
- Object-Oriented Programming (Wikipedia)
  - Self-contained Python class for each topic
  - Classes can be divided and worked on in separate files (and work divided among multiple collaborators)
  - Class `__init__` function instances the class and documents the attributes (aka variables) for the work
  - Single-action class functions (aka “methods”) do the work
  - Jupyter procedural cells or class `@property` functions instance the classes and call functions to do the model’s work
- Test-Driven Development (Wikipedia)
  - Use Pytest and test datasets to validate your code (coding consists of creating parallel `test_xxx.py` and `xxx.py` files)
  - Pre-write the test; Write simplest version of function that passes; Refactor to improve the code

EXAMPLE MODEL FOR LEARNING ”TUHDOOP”

- Model is a teaching case study in Digital Transformation with Excel Online Course
- We can use its raw and transformed data for validating a Python version
- Roll Length Model.xlsx - Toilet paper example
  - Technical model to calculate length of substrate on a roll based on diameter
  - Model is industrially useful in R&D and manufacturing settings
  - Example raw data and equation derivation available as background in file

$$L = \frac{\pi({D_{roll}}^2 - {D_{core}}^2)}{4C} \Rightarrow L = \left(\frac{\pi}{4C}\right){D_{roll}}^2 + \left(\frac{-\pi {D_{core}}^2}{4C}\right) \Rightarrow Y = mX + b$$

Transformed/Linearized



OBJECT-ORIENTED PYTHON BASICS

- Classes (aka objects) have an __init__ method to populate attributes that are inputs to a model or data transformation
- __init__ runs whenever the class is instanced with a statement like this:

```
r = RollLength(file_raw='raw_data.xlsx')
```

- Good practice is initialize non-input attributes to **None** or empty string in __init__
- Making some or all inputs optional is useful for using the Class for multiple use cases requiring different inputs
- self is placeholder referring to current instance. Think of it as bucket carrying current values of all attribute variables (and as required prefix for calling methods)
- Refer to a class attribute as self.var_name

```
class RollLength:
    def __init__(self, file_raw='', diam_roll=None, diam_core=None, caliper=None):
        """
        Initializes a RollLength object
        JDL 4/27/23

        Args:
        file_raw (string, optional): Directory path + filename for raw, length versus diam data
            where rows represent measurements on a roll as material is unwound.
        diam_core (float, optional): Diameter in mm of the core that the material is wound onto.
        diam_roll (float, optional): Roll diameter [mm] for the substrate roll
        caliper (float, optional): Thickness of the substrate measured in mm.
        """

        #CalculateRollLength procedure
        self.diam_core = diam_core
        self.diam_roll = diam_roll
        self.caliper = caliper
        self.length = None #Substrate roll length [m] Calculated by .CalculateLength()

        #CaliperFromRawData procedure attributes
        self.file_raw = file_raw
        self.df_raw = None #DataFrame with raw length [m] versus diameter [mm] exptl. data
        self.slope = None #Calculated slope from linear fit
        self.intercept = None #Calculated y-intercept from linear fit
        self.R_squared = None #Calculated R-Squared from linear fit
```

OBJECT-ORIENTED BASICS (CONTINUED)

- Method functions have **self** as an argument to bring in current values of all attributes
  - In the example, **self.df_raw** is a needed input
  - The method modifies **self.df_raw** by adding two columns
- Methods can also have other arguments that are not class attributes –as needed to bring values within the method's scope for calculations
- Methods do not need to have a **return** statement if modifying class attributes
- After the method runs, the modified **df_raw** will be the current value in the instance of **RollLength**

Example RollLength Class Method

```
def AddCalculatedRawCols(self):
    """
    Add Calculated columns to length, diam raw measurement data
    """
    self.df_raw['diam_m'] = self.df_raw['diameter'] / 1000
    self.df_raw['diam_m^2'] = self.df_raw['diam_m'] ** 2
```

OBJECT-ORIENTED BASICS (CONTINUED)

- Python classes can be created in a Jupyter notebook but generally should put them in a separate *.py file
  - makes user-facing code cleaner / gobbledygook reduction for users
  - critical for giving Pytest access to code for validation
- from roll import RollLength statement makes the class accessible from a Jupyter notebook (or other *.py file such as the test_xxx.py files)
- The *.py file's location needs to be added to sys.path, which is a list of locations Python will look in for importing

File structure for using a user-facing Jupyter Notebook Roll_Length_Level3.ipynb

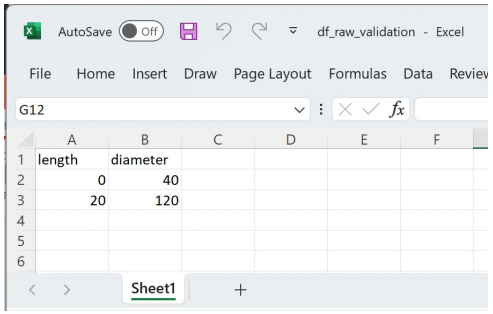
```
Roll_Length_Model
    ├── Roll_Length_Level3.ipynb   User-facing Jupyter Notebook
    ├── cushiony_tp_length_vs_diam.xlsx   Input file (production)
    ├── libs
    │   └── roll.py   File containing RollLength Class code
    └── tests
        ├── df_raw_validation.xlsx   Input file (for validation)
        └── test_roll.py   Pytest test code
```

sys.path addition of libs folder and import RollLength Class

```
#Orient the notebook to its location and import Rolllength Class from roll.py
path_libs = os.getcwd() + os.sep + 'libs'
if path_libs not in sys.path: sys.path.append(path_libs)
from roll import RollLength
```

TDD / PYTEST BASICS

```
def test_ReadRawData():
    """
    Import experimental length versus diam data to Pandas DataFrame
    """
    roll_raw_fit = RollLength(file_raw:='df_raw_validation.xlsx')
    roll_raw_fit.ReadRawData()
    assert roll_raw_fit.df_raw.index.size == 2
    assert roll_raw_fit.df_raw.loc[1, 'length'] == 20
```



	A	B	C	D	E	F
1	length	diameter				
2	0	40				
3	20	120				
4						
5						
6						

- The standard Pytest library allows setting up one or more parallel test code files for validation
- Run a test file with this statement (runs all test functions whose names begin with 'test_' prefix

```
$python –m pytest test_roll.py –v -s
```

- Run an individual test by adding –k modifier

```
$python –m pytest –v –s –k 'test_ReadRawData'
```

- Pytest evaluates results based on assert statements you add. If all are True, the test passes
  - right side of assert needs to evaluate as True (Pass) or False (Fail)
  - Remember to use "==" for testing equality

TDD / PYTEST BASICS (CONTINUED)

```
@pytest.fixture()
def roll_raw_fit():
    return RollLength(file_raw:='df_raw_validation.xlsx')

"""
========================================================================
CaliperFromRawData Procedure
========================================================================
"""
def test_ReadRawData(roll_raw_fit):
    """
    Import experimental length versus diam data to Pandas DataFrame
    """
    roll_raw_fit.ReadRawData()
    assert roll_raw_fit.df_raw.index.size == 2
    assert roll_raw_fit.df_raw.loc[1, 'length'] == 20
```

- The @pytest.fixture decorator is useful for populating items that are needed for multiple tests
- The roll_raw_fit Class instance is an example
- The fixture will consist of whatever is in the return statement when it is created
- Include the fixture as argument in tests that need it
- Fixtures are re-evaluated for each test, so it's ok if a test modifies a fixture

Compare this version of the test with the previous slide that does not use a fixture

CODE ARCHITECTURE

- In our context, architecture is the macroscopic arrangement of application and test code (e.g. not the fine, Python style details)
- Good architecture is a key for curation, validation and maintainability
- Useful to classify functions as either multi-step “procedures” or “single-action”
  - Single-action functions are the preferred elemental building block of coded projects
  - Procedures walk through a recipe of steps to complete a use case –typically by sequentially calling the single-action functions

Example procedure function that calls a recipe of four single-action functions

```
def CaliperFromRawDataProcedure(self):
    """
    Procedure to fit a line to transformed raw, length versus diam data
    and thereby enable calculation of an effective caliper for the
    material on a roll of substrate.

    This use case only uses the file_raw Class input --to read in raw
    data
    """
    self.ReadRawData()
    self.AddCalculatedRawCols()
    self.FitRawData()
    self.CalculateCaliper()
```

Example single-action function that adds two, calculated columns to the df_raw DataFrame

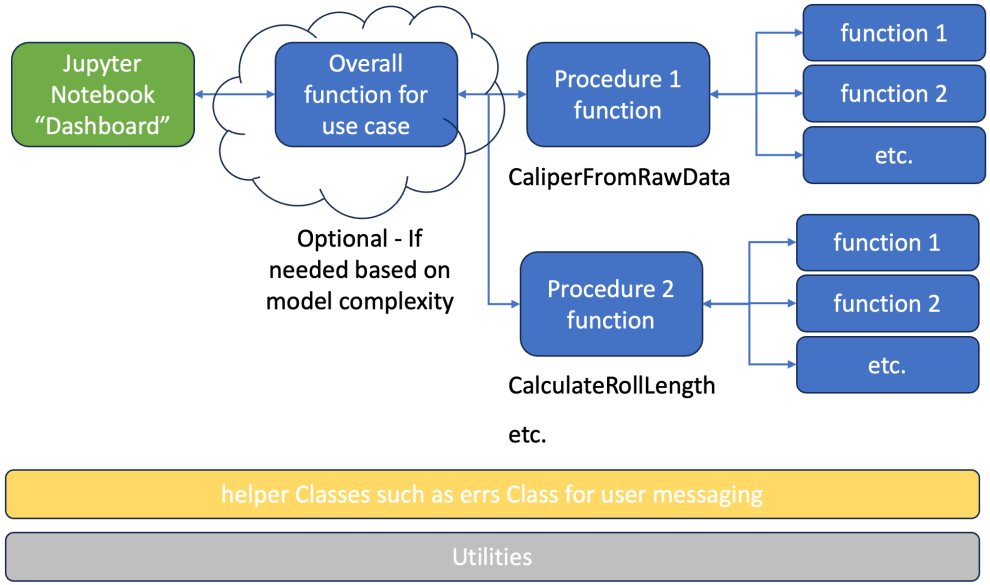
```
def AddCalculatedRawCols(self):
    """
    Add Calculated columns to length, diam raw measurement data
    """
    self.df_raw['diam_m'] = self.df_raw['diameter'] / 1000
    self.df_raw['diam_m^2'] = self.df_raw['diam_m'] ** 2
```

TYPICAL CODING STYLES PEOPLE USE FOR PYTHON MODELS AND DATA ANALYSIS

- Level 1 – ad hoc “random walk” Jupyter notebook
  - Fast
  - Not well curated
  - Hard to validate with re-runnable checks
  - Hard to maintain/extend
- Level 2 – Procedure/function hierarchies Jupyter
  - Better curation and extensibility
  - Better for validation but not Pytest-compatible
- Level 3 – TDD/OOP aka Tuhdoop
  - Ideal for models that will be re-used
  - Conducive to re-runnable Pytest testing
  - Use Jupyter notebook as UI only; put most code in *.py file(s)

SUGGESTED ARCHITECTURE

- A good, general-purpose architecture is 4-level structure
- Right side blocks are single-action functions (ok to add sub-level for DRY aka avoid repitition)
- Overall function allows single-line running from Jupyter
- Procedure functions document the sequence needed to execute
- This applies to either Level 2 or Level 3 coding styles



SUGGESTED LEVEL 3 CODE STYLE

- For curation, arrange code and tests in hierarchy corresponding to block diagram
- Can be multiple tests per function but at least 1-for-1
- Use @property function to return key procedure output(s) with single line statement
- tests typically run/validate single-action functions but ok to include tests of procedures too

SUGGESTED LEVEL 3 CODE STYLE

- Use a Jupyter notebook to create a user interface dashboard for running *.py code
- Procedure functions serve as wrappers for single-action functions
  - single-line running for @property
  - two-line instance + run for procedure functions

Roll Length "Level 3" TDD/OOP Model

- This notebook is a user interface for a model whose calculations are mostly contained in another code file (roll.py) in object-oriented style. The model is validated by a separate test_roll.py file containing Pytest tests.
- For a substrate on a roll such as toilet paper or an industrial substrate, this notebook can be used to fit experimental length versus diameter data to calculate the substrate's effective caliper while wound on the roll
- Alternatively, if caliper and diameter are known from previous measurements, it can calculate roll length. This is useful industrially to know how many linear meters remain on a roll and thereby be able to calculate parameters like roll weight and run time.

JDL / DataDelve LLC, April 2023; Updated July 2023

[1]:
```
import pandas as pd
import sys, os

#Orient the notebook to its location and import Rolllength Class from roll.py
path_scripts = os.sep.join(os.getcwd().split(os.sep)[:-2] + ['roll_scripts'])
if path_scripts not in sys.path: sys.path.append(path_scripts)
from roll import RollLength
```
Import the model's Class from roll.py

[2]:
```
# Unit Conversions Used by Model
d_unit_conv = {'mm_m':1000.,
               'g_kg':1000.,
               's_min':60}
```

How to use the RollLength class

- If all we want is to calculate caliper, the class contains a Python `@Property` called `CaliperFromRawData` to do this
- The class also contains methods for plotting data. To use these, we need to first instance the class with the raw data as an input

Call an @property Class function to compute a result in a single step

[3]:
```
#Using the property to directly return caliper
print(RollLength(file_raw='cushiony_tp_length_vs_diam.xlsx').CaliperFromRawData, 'is the calculated caliper in mm')
```
```
0.4804 is the calculated caliper in mm
```

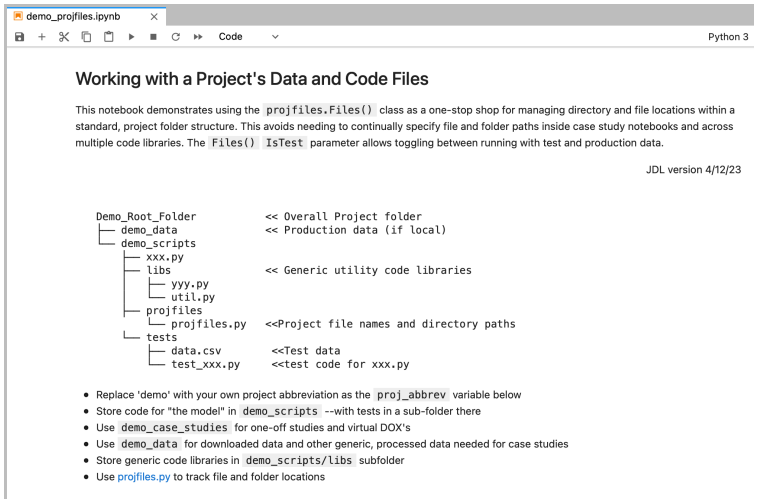
[4]:
```
#Instance the class and calculate caliper
r = RollLength(file_raw='cushiony_tp_length_vs_diam.xlsx')
r.CaliperFromRawDataProcedure()
```
Instance the RollLength class and run a procedure to import data, transform/fit data and calculate caliper

```
# Once the "procedure" has been run, print calculated attributes of instanced class
print('R^2: ', round(r.R_squared, 4), 'Caliper: ', r.caliper)
#print('\n', r.df_raw)
```
Print some procedure outputs
```
R^2:  0.9985 Caliper:  0.4804
```

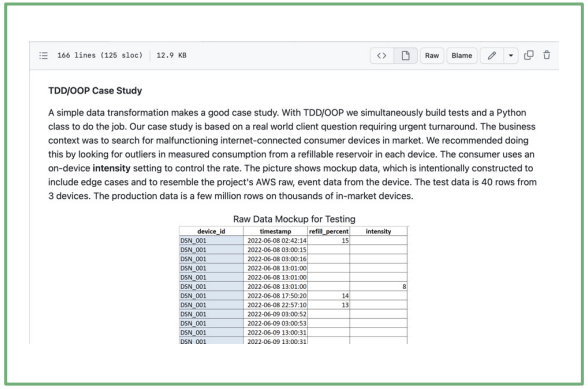
[5]:
```
#The class includes a plotting procedure
r.PlotRawAndTransformedData()
```
Run a procedure to plot the data generated within the class instance

KEEPING TRACK OF A PROJECT'S FILES AND CODE

- Open Jupyter Notebook
  `Demo_Root_Folder/demo_analysis/_Dev_ThisProject/demo_projfiles.ipynb`
- The folder structure shows how to set things up for the model to be developed in a central place (demo_scripts)
- **`_Dev_ThisProject`** folder represents the developer's latest version. It has a Jupyter notebook as "dashboard" UI for running the model
- Users can copy (and rename!) the **`_Dev_ThisProject`** folder and have instant access to latest version and connection to all data and scripts



GETTING STARTED WITH PYTHON CLASSES AND TESTING



- Homework for next time:
  - Download repository and review tutorial.md from TDD_OOP_Tutorial Github repo
  - Great Twitter thread by Stephen Gruppetta teaching Object-Oriented Programming.
    - Entitled, “Object-Oriented Python at the Hogwarts School of Codecraft and Algorithmancy” (he picked a theme and stuck with it…)
    - See his “Years” 1, 2 and 3 threads (4 through 7 are great but mostly beyond scope of what we will need and cover in next session)
      - Year 1
      - Year 2
      - Year 3