DESIGNSAFE-CI

A NATURAL HAZARDS ENGINEERING COMMUNITY

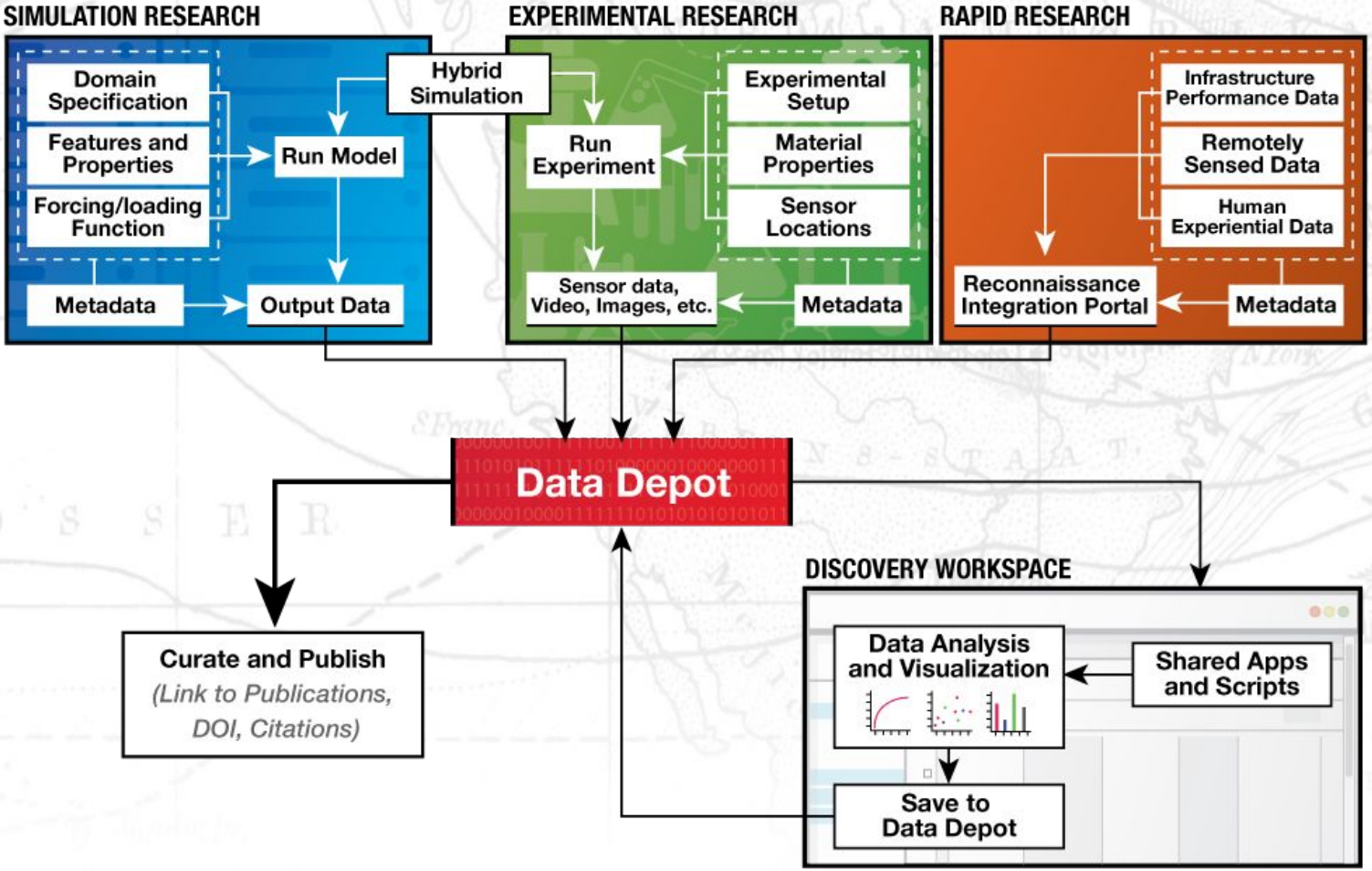
A New Cyberinfrastructure for the Natural Hazards Community

DESIGNSAFE-CI A NATURAL HAZARDS ENGINEERING COMMUNITY | TEXAS The University of Texas at Austin | TACC | RICE | Florida Tech

DesignSafe Vision

- A CI that is an integral and dynamic part of research discovery
- Cloud-based tools that support the analysis, visualization, and integration of diverse data types
  - Key to unlocking the power of “big data”
- Support end-to-end research workflows and the full research lifecycle
- Enhance, amplify, and link the capabilities of the other NHERI components

DesignSafe: Enabling Research



Agenda

- Introduction to Slack
- Introduction to the Jupyter Notebook
- Using Numpy
- Using Pandas
- Animation using Matplotlib
- Interactive Plots using MPLD3
- Launching DesignSafe Applications using AgavePy

What's Slack?

In a nutshell, Slack is a communication tool. But, it's a bit more than that.

The Lingo:

- Teams
  - Slack is divided into teams, it's designate in the url you connect to.
  - https://designsafe-ci.slack.com/
- Channels
  - These are "Topics of Discussion"
  - Anyone can create a channel
- Threads:
  - You can group communications together and create subtopics

What's Slack?

But it's more than just a "Communication Tool"

*after the presentation is over, we'll continue the conversation on slack, https://designsafe-ci.slack.com/

What are Jupyter Notebooks?

A web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

How do Jupyter Notebooks Work?

An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel.

What's this mean?

Notebooks are an interface to kernel, the kernel executes your code and outputs back to you through the notebook. The kernel is essentially our programming language we wish to interface with.

Jupyter Notebooks, Structure

- Code Cells
  - Code cells allow you to enter and run code
    Run a code cell using Shift-Enter

- Markdown Cells
  - Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML.

Jupyter Notebooks, Structure

- Markdown Cells
  - You can add headings:
    - # Heading 1
      # Heading 2
      ## Heading 2.1
      ## Heading 2.2
  - You can add lists
    - 1. First ordered list item
      2. Another item
      · · * Unordered sub-list.
      1. Actual numbers don't matter, just that it's a number
      · · 1. Ordered sub-list
      4. And another item.

Jupyter Notebooks, Structure

- Markdown Cells
  - pure HTML
    - <dl>
      <dt>Definition list</dt>
      <dd>Is something people use sometimes.</dd>

      <dt>Markdown in HTML</dt>
      <dd>Does *not* work **very** well. Use HTML tags.</dd>
      </dl>
  - And even, Latex!
    - $e^{i\pi} + 1 = 0$

Jupyter Notebooks, Workflow

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

Jupyter Notebooks, Workflow

- Let a traditional paper lab notebook be your guide:
  - Each notebook keeps a historical (and dated) record of the analysis as it's being explored.
  - The notebook is not meant to be anything other than a place for experimentation and development.
  - Notebooks can be split when they get too long.
  - Notebooks can be split by topic, if it makes sense.

Jupyter Notebooks, Shortcuts

- `Shift-Enter`: **run cell**
  - Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter`is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.

Jupyter Notebooks, Shortcuts

- `Ctrl-Enter`: **run cell in-place**
  - Execute the current cell as if it were in “terminal mode”, where any output is shown, but the cursor *remains* in the current cell. The cell’s entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

Jupyter Notebooks, Shortcuts

- `Alt-Enter`: **run cell, insert below**
  - Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). (shortcut for the sequence `Shift-Enter`,`Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.))
- `Esc` **and** `Enter`: **Command mode and edit mode**
  - In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

Python - NumPy

- "Numerical Python"
- open source extension module for Python
- provides fast precompiled functions for mathematical and numerical routines
- adds powerful data structures for efficient computation of multi-dimensional arrays and matrices.

NumPy, First Steps

Let build a simple list, turn it into a numpy array and perform some simple math.

```
import numpy as np
cvalues = [25.3, 24.8, 26.9, 23.9]
C = np.array(cvalues)
print(C)
```

NumPy, First Steps

Let build a simple list, turn it into a numpy array and perform some simple math.

```
print(C * 9 / 5 + 32)
```

vs.

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

NumPy, Cooler things

```
import time
size_of_vec = 1000
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

NumPy, Cooler things

Let's see which is faster.

```
t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
```

NumPy, Multi-Dimension Arrays

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)


B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
print(B.ndim)
```

NumPy, Multi-Dimension Arrays

The shape function:

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))
```

NumPy, Multi-Dimension Arrays

The shape function can also *change* the shape:

```
x.shape = (3, 6)
print(x)

x.shape = (2, 9)
print(x)
```

NumPy, Multi-Dimension Arrays

A couple more examples of shape:

```
x = np.array(42)
print(np.shape(x))

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B.shape)
```

NumPy, Multi-Dimension Arrays

indexing:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])

# print the first element of F, i.e. the element with the index 0
print(F[0])

# print the last element of F
print(F[-1])

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B[0][1][0])
```

NumPy, Multi-Dimension Arrays

slicing:

```
A = np.array([
[11,12,13,14,15],
[21,22,23,24,25],
[31,32,33,34,35],
[41,42,43,44,45],
[51,52,53,54,55]])

print(A[:3,2:])


print(A[3:,:])
```

NumPy, Multi-Dimension Arrays

identity function

```
np.identity(4)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def TimeStep(self, dt=0.0):
    """Takes a time step using straight forward Python loops."""
    g = self.grid
    nx, ny = g.u.shape
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    err = 0.0
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            tmp = u[i,j]
            u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                      (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
            diff = u[i,j] - tmp
            err += diff*diff

    return numpy.sqrt(err)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def numericTimeStep(self, dt=0.0):
    """Takes a time step using a NumPy expression."""
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    g.old_u = u.copy() # needed to compute the error.

    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

    return g.computeError()
```

Pandas, What is it?

A software library written for the Python for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
s = pd.Series([1,3,5,np.nan,6,8])
s
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
dates = pd.date_range('20130101',
 periods=6)
dates
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
df = pd.DataFrame(np.random.randn(6,4),
 index=dates, columns=list('ABCD'))
df
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
df = pd.DataFrame(np.random.randn(6,4),
 index=dates, columns=list('ABCD'))
df
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
df2 = pd.DataFrame({ 'A' : 1.,'B' :
 pd.Timestamp('20130102'),'C' :
 pd.Series(1,index=list(range(4)),dtype='float32'),'D' :
 np.array([3] * 4,dtype='int32'),'E' :
 pd.Categorical(["test","train","test","train"]),'F' :
 'foo' })

df2
```

Pandas, Viewing Data

Some common/useful functions

```
df.head()
df.tail(3)
df.index
df.columns
df.values
df.describe()
df.T
df.sort_index(axis=1, ascending=False)
df.sort_values(by='B')
```

Pandas, Selecting Data by Label

Some common/useful functions

```
df['A'])
df[0:3]
df['20130102':'20130104']
df.loc[dates[0]]
df.loc[:,['A','B']]
df.loc['20130102':'20130104',['A','B']]
df.loc['20130102',['A','B']]
df.loc[dates[0],'A']
```

Pandas, Selecting Data by Position

Some common/useful functions

```
df.iloc[3]
df.iloc[3:5,0:2]
df.iloc[[1,2,4],[0,2]]
df.iloc[1:3,:]
df.iloc[:,1:3]
df.iloc[1,1]
df.iat[1,1]
```

Pandas, Summary of Features

Pandas allow for:

- Boolean Indexing
- Statistical Operations
- Histogramming
- Merging Data
- SQL Style Joins
- SQL Style Appends
- SQL Style Grouping
- Reshaping
- Pivoting
- and more!

Pandas, CSV Files

manipulating CSV files.

```
ts = pd.Series(np.random.randn(1000),
 index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()

df = pd.DataFrame(np.random.randn(1000, 4),
 index=ts.index,columns=['A', 'B', 'C', 'D'])
df = df.cumsum()

df.to_csv('foo.csv')
pd.read_csv('foo.csv')
```

Matplotlib, What is it?

It's a graphing library for Python. It has a nice collection of tools that you can use to create anything from simple graphs, to scatter plots, to 3D graphs. It is used heavily in the scientific Python community for data visualisation.

Matplotlib, First Steps

Let's plot a simple sin wave from 0 to 2 pi.

First lets, get our code started by importing the necessary modules.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Matplotlib, First Steps

Let's add the following lines, we're setting up x as an array of 50 elements going from 0 to 2*pi

```
x = np.linspace(0, 2 * np.pi, 50)
plt.plot(x, np.sin(x))
plt.show() # Show the graph.
```

Let's run our cell!

Matplotlib, a bit more interesting

Let's plot another curve on the axis

```
plt.plot(x, np.sin(x),
         x, np.sin(2 * x))
plt.show()
```

Let's run our cell!

Matplotlib, a bit more interesting

Let's see if we can make the plots easier to read

```
plt.plot(x, np.sin(x), 'r-o',
         x, np.cos(x), 'g--')
plt.show()
```

Let's run this cell!

Matplotlib, a bit more interesting

```
Colors:
Blue – ‘b’
Green – ‘g’
Red – ‘r’
Cyan – ‘c’
Magenta – ‘m’
Yellow – ‘y’
Black – ‘k’ (‘b’ is taken by blue so the
last letter is used)
White  – ‘w’
```

Matplotlib, a bit more interesting

```
Lines:
Solid Line – ‘-‘
Dashed – ‘–‘
Dotted – ‘.’
Dash-dotted – ‘-:’

Often Used Markers:
Point – ‘.’
Pixel – ‘,’
Circle – ‘o’
Square – ‘s’
Triangle – ‘^’
```

Matplotlib, Subplots

Let's split the plots up into subplots

```
plt.subplot(2, 1, 1) # (row, column, active
area)
plt.plot(x, np.sin(x), 'r')
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x), 'g')
plt.show()
```

using the subplot() function, we can plot two graphs at the same time within the same "canvas". Think of the subplots as "tables", each subplot is set with the number of rows, the number of columns, and the active area, the active areas are numbered left to right, then up to down.

Matplotlib, Scatter Plots

Let's take our sin curve, and make it a scatter plot

```
y = np.sin(x)
plt.scatter(x,y)
plt.show()
```

call the scatter() function and pass it two arrays of x and y coordinates.

Matplotlib, add a touch of color

Let's mix things up, using random numbers and add a colormap to a scatter plot

```
x = np.random.rand(1000)
y = np.random.rand(1000)
size = np.random.rand(1000) * 50
color = np.random.rand(1000)
plt.scatter(x, y, size, color)
plt.colorbar()
plt.show()
```

Matplotlib, add a touch of color

Let's see what we added, and where that takes us

```
...
plt.scatter(x, y, size, color)
plt.colorbar()
...
```

We brought in two new parameters, size and color. Which will varies the diameter and the color of our points. Then adding the colorbar() gives us nice color legend to the side.

Matplotlib, Histograms

A histogram is one of the simplest types of graphs to plot in Matplotlib. All you need to do is pass the hist() function an array of data. The second argument specifies the amount of bins to use. Bins are intervals of values that our data will fall into. The more bins, the more bars.

```
plt.hist(x, 50)
plt.show()
```

Matplotlib, Contour Plots

Let's play with our preloaded data.

```
import matplotlib.cm as cm

with open('../mydata/ContourData/contourData.dat') as file:
    array2d = [[float(digit) for digit in line.split()] for line in file]
print array2d

nx, ny = np.shape(array2d)

cs = plt.pcolor(array2d, cmap=cm.get_cmap('afmhot'))
cb = plt.colorbar(cs, orientation = 'horizontal')
plt.xlim(0,nx)
plt.ylim(0,ny)
plt.show()
```

Matplotlib, Adding Labels and Legends

Let's go back to our sin/cos curve example, and add a bit of clarification to our plots

```
x = np.linspace(0, 2 * np.pi, 50)
plt.plot(x, np.sin(x), 'r-x', label='Sin(x)')
plt.plot(x, np.cos(x), 'g-^', label='Cos(x)')
plt.legend() # Display the legend.
plt.xlabel('Rads') # Add a label to the x-axis.
plt.ylabel('Amplitude') # Add a label to the
y-axis.
plt.title('Sin and Cos Waves') # Add a graph
title.
plt.show()
```

Matplotlib, Animating

animation.FuncAnimation(...)
Makes an animation by repeatedly calling a function func.

class matplotlib.animation.FuncAnimation(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, ***kwargs*)

Matplotlib, Animating

```
%pylab inline
from matplotlib import animation

# First set up the figure, the axis, and
the plot element we want to animate
fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2))
line, = ax.plot([], [], lw=2)

# initialization function: plot the
background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called
sequentially
def animate(i):
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,
```

```
# call the animator.  blit=True means only
re-draw the parts that have changed.
anim = animation.FuncAnimation(fig,
animate, init_func=init, frames=100,
interval=20, blit=True)

# call our new function to display the
animation
display_animation(anim)
```

Matplotlib, Animating

```
from IPython.display import HTML

def display_animation(anim):
    plt.close(anim._fig)
    return HTML(anim_to_html(anim))
```

Matplotlib, Animating

```
from tempfile import NamedTemporaryFile

VIDEO_TAG = """<video controls>
 <source src="data:video/x-m4v; base64,{0}" type="video/mp4">
 Your browser does not support the video tag.
</video>"""

def anim_to_html(anim):
    if not hasattr(anim, '_encoded_video'):
        with NamedTemporaryFile(suffix='.mp4') as f:
            anim.save(f.name, fps=20, extra_args=['-vcodec', 'libx264'])
            video = open(f.name, "rb").read()
        anim._encoded_video = video.encode("base64")

    return VIDEO_TAG.format(anim._encoded_video)
```

mpld3, What is it?

An API that merges Matplotlib, the popular Python-based graphing library, and D3js, a popular JavaScript library for creating interactive data visualizations for the web.

mpld3, First Steps

customizing your Jupyter session.

```
pip install --user mpld3
```

mpld3, Demo

http://mpld3.github.io/index.html

The Agave API, What is it?

Agave is an open source, platform-as-a-service solution for hybrid cloud computing. It provides a full suite of services covering everything from standards-based authentication and authorization to computational, data, and collaborative services.

The Agave API, What is it?

DesignSafe uses Agave to interact with the "behind the scenes" resources.

We can use Agave to launch our DesignSafe applications from Jupyter.

Agavepy, By Example

First, we need to import the Agave class from the agavepy package.

```
from agavepy.agave import Agave
```

Agavepy, By Example

With the Agave class imported, we can now instantiate our client. Typically, this would involve passing your OAuth credentials, but because we are in a notebook on JupyterHub, we can use the "restore" shortcut to create a client with credentials already saved for us behind the scenes.

```
ag = Agave.restore()
```

Agavepy, By Example

We need our application ID

```
ag.apps.list()
```

Agavepy, By Example

Then go through the steps to launch our app

```
app_id = 'opensees-2.4.4.5804'
app = ag.apps.get(appId=app_id)
```

Agavepy, By Example

- giving the application inputs.
- parameters
- the job description
- then submit

see:

https://github.com/TACC/jupyterhub_images/blob/master/designsafe/opensees-submit-example.ipynb

Summary

- Using Numpy
  - build our data
- Using Pandas
  - analyze our data
- Animation using Matplotlib
  - view our data
- Interactive Plots using MPLD3
  - play with our data
- Launching DesignSafe Applications using AgavePy
  - launch your app

DesignSafe: Questions?

For additional questions, feel free to contact us

- Slack:
  https://designsafe-ci.slack.com

- Email:
  training@designsafe-ci.org

- or fill out a ticket:
  https://www.designsafe-ci.org/help/tickets

DesignSafe: Thanks

Ellen Rathje, Tim Cockerill, Jamie Padgett, Dan Stanzione, Steve Mock, Joe Stubbs, Josue Coronel, Craig Jansen, Matt Stelmaszek, Hedda Prochaska, Joonyee Chuah

DesignSafe: References

- http://www.datadependence.com/2016/04/scientific-python-matplotlib/
- http://jupyter-notebook.readthedocs.io/en/latest/notebook.html
- http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html
- https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
- https://www.codecademy.com/learn/python
- http://scipy-cookbook.readthedocs.io/items/PerformancePython.html
- http://www.python-course.eu/numpy.php
- http://pandas.pydata.org/pandas-docs/stable/10min.html
- http://mpld3.github.io/index.html
-

DesignSafe: Next Webinar

DesignSafe:

Interactive Plots and Data Analysis with R

Wednesday, March 29th 1:00p – 3:00p CST