**NHRAP Docstring Conventions**

Docstring conventions are documentation standards for classes, methods and functions within code. It is important to have standard conventions to ease collaboration in development and debugging. For purposes of development for the NHRAP, I am advocating we follow a modified version of the PEP 257 Docstring Conventions (<https://www.python.org/dev/peps/pep-0257/>).

**What is a docstring**

A docstring is a multi-line comment documenting a class, method or function. In the Python example below, it begins and ends with three double quotations (“””). The comment syntax will vary across programing languages, but regardless, docstrings should be comprised of three parts.

**Three parts of a docstring**

1. An explanation of what the class, method or function does
   1. This should be generally for context and purpose
2. Keyword arguments: what is being passed into the class, method or function
   1. Keyword arguments should have three parts, the name of the argument, the data type of the argument, and a brief explanation. If the argument is complicated it can be useful to have an example.

argument\_variable\_name: data\_type -- explanation and notes of argument (example)

* 1. If the data type is a list or array, you can specify the sub data type as below

argument\_variable\_name: data\_type<sub\_data\_type> -- explanation (example)

1. Returns: anything that is returned by the method or function
   1. Returns are constructed in the same way as arguments

return\_variable\_name: data\_type – explanation and notes (example)

**Examples**

Simple example:

def complex(real=0.0, imag=0.0):

"""Form a complex number.

Keyword arguments:

real: float -- the real part (default 0.0)

imag: float -- the imaginary part (default 0.0)

Returns:

complex\_zero: float – a complex number

"""

...

Working example:

Link: <https://github.com/nhrap-dev/windgrid-dat/blob/master/windgrid-dat.py>

def idw(kdtree,z,xi,yi):

""" Inverse Distance Weighting - interpolates an unknown value at a

specified point by weighting the values of it's nearest neighbors

Keyword arguments:

kdtree: scipy.spatial.ckdtree -- kdtree made from a 2d array of x and y coordinates as the columns

z: 1d array -- point values at each location

xi: float -- x-axis point location of unknown value

yi: float -- y-axis point location of unknown value

Returns:

zi: float -- interpolated value at xi, yi

"""

neighbors = 12

power = 2

distances, indicies = kdtree.query([xi,yi], k=neighbors)

z\_n = z[indicies]

if 0 not in distances:

weights = power / distances

else:

distances += 0.000000001

weights = power / distances

weights /= weights.sum(axis=0)

zi = np.dot(weights.T, z\_n)

return zi

If returns or keyword arguments don’t exist, they don’t need to be added:

def drop\_problem\_tables(cnxn):

""" Drops user defined tables in database that may cause problems

Keyword arguments:

cnxn: pyodbc connection -- connection to Hazus database

"""

# tables to drop

sql\_query\_drop = ["DROP TABLE dbo.huUserDefinedFlty;",

"DROP TABLE dbo.eqUserDefinedFlty",

"DROP TABLE dbo.flUserDefinedFlty",

"DROP TABLE dbo.tsUserDefinedFlty"]

# drop problem tables

for query in sql\_query\_drop:

try:

cursor = cnxn.cursor()

cursor.execute(query)

cursor.commit()

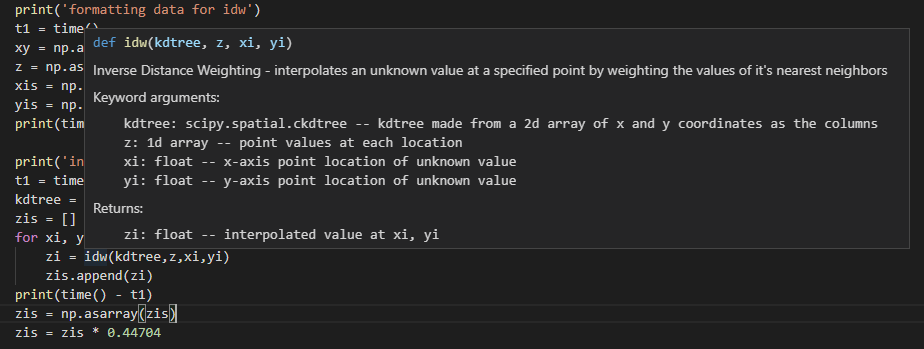
print(query + ' successful')

except Exception as e:

print(str(e))

**Docstring inspection**

In advanced text editors and IDEs, the docstrings will be available if you inspect or hover over your class, method or function and can be very useful in executing and debugging code without having to return to the implementation. Below, hovering over the function idw in VSCode, I can view the docstring.



**Inline documentation**

It is important to document within the classes, methods and functions in addition to docstrings. No conventions are advocated at this time, but general explanation and is encouraged for collaboration and debugging purposes. Inline documentation can be constructed in multiple ways.

* As preceding comments

# this code does a thing

module.thing()

* As inline comments

module = Module() #initializes module

module.properties = [x.value for x in list] # prepares the module data

module.thing() # creates the thing

* As print statements

print(“intializing module and running thing”)

module = Module()

module.properties = [x.value for x in list]

module.thing()

Example

def create\_dat(windgrid\_file, DAT\_header, output\_file):

""" Creates a Hazus DAT file containing windspeeds in m/s from a windgrid shapefile

Keyword arguments:

windgrid\_file: str -- file location of windgrid shapefile

DAT\_header: list<str> -- a list of strings to be used as the header of the DAT file

output\_file: str -- file location and name of output DAT file

"""

# read windgrid

output\_file = output\_file + '.dat'

windgrid = gpd.read\_file(windgrid\_file)

# read centroids

centroids\_all = gpd.read\_file('base\_data/us\_centroids.shp')

# select centroids in windgrid

buff = windgrid.geometry.buffer(0.2)

buff\_gdf = gpd.GeoDataFrame(geometry=buff.geometry)

buff\_gdf['dis'] = 1

dissolve = buff\_gdf.dissolve(by='dis')

centroids\_intersect = centroids\_all.intersects(dissolve.unary\_union)

centroids = centroids\_all[centroids\_intersect == True]

# format data for idw

t1 = time()

xy = np.asarray([[x.x, x.y] for x in windgrid.geometry])

z = np.asarray([x for x in windgrid.Vg\_mph])

xis = np.asarray([x.x for x in centroids.geometry])

yis = np.asarray([x.y for x in centroids.geometry])

# interpolate values

kdtree = cKDTree(xy)

zis = []

for xi, yi in zip(xis, yis):

zi = idw(kdtree,z,xi,yi)

zis.append(zi)

zis = np.asarray(zis)

zis = zis \* 0.44704

# format dataframe for output

tracts = list(map(lambda x: x + ' ', centroids.FIPS))

longs = list(map(lambda x: '{0:.4f}'.format(x) + ' ', xis))

lats = list(map(lambda x: '{0:.4f}'.format(x) + ' ', yis))

windSpeeds = list(map(lambda x: '{0:.5f}'.format(x) + ' ', zis))

zeros = list(map(lambda x: '0' + '{0:.5f}'.format(x \* 0) + ' ', zis))

windSpeedsLast = list(map(lambda x: '{0:.5f}'.format(x), zis))

df = pd.DataFrame({'tracts': tracts, 'longs': longs, 'lats': lats, 'windSpeeds': windSpeeds, 'zeros': zeros, 'windSpeedsLast': windSpeedsLast})

# write output DAT file

# creates and opens the export DAT file

pd.DataFrame().to\_csv(output\_file, header=False, index=False)

export=open(output\_file, "w")

# add columns to the DAT file header

DAT\_header.append('')

DAT\_header.append(' ident elon nlat ux vy w (m/s)')

# write header to DAT file

for row in DAT\_header:

export.write(row + '\n')

# write data to DAT file

for row in range(len(df[df.columns[0]])):

writeRow = ''

for item in df.iloc[row]:

writeRow = writeRow + item

export.write(writeRow + '\n')

export.close()