Python Data Models
in ArcGIS

Matthew J. Yoder
Champaign County Regional Planning Commission

ILGISA Annual Conference
September 15, 2015

CHAMPAIGN COUNTY
rpc
REGIONAL
PLANNING
COMMISSION

0 TERMINOLOGY

cuuats.datamodel

Champaign Urbana
Urbanized Area
Transportation Study

A data access layer that includes schema, business logic and relationships

- Open-source (BSD license)
- Available on GitHub: https://github.com/cuuats
- Generally useful for GIS tasks (not specific to CUUATS)
- Distributed as a Python egg (install using pip)

0 AGENDA

1 An example
2 Why use a data model?
3 Case study
4 Building a Python data model
5 Querying the model
6 Validating data
7 Calculating values
8 Working with relationships
9 Next steps

1 AN EXAMPLE

From a feature class of bus stops, select features where the shelter type is "covered," and save the attached photo in the format {StopID}.jpg.

BusStop

- OBJECTID (OID)
- GlobalID (GlobalID)
- SHAPE (Point geometry)
- StopID (Short integer)
- StopName (Text)
- RouteNumber (Short integer)
- ShelterType (Short integer)

Coded Values Domain:
1: Open
2: Covered
100: None

BusStop__ATTACH

- ATTACHMENTID (OID)
- REL_GLOBALID (GUID)
- CONTENT_TYPE (Text)
- ATT_NAME (Text)
- DATA_SIZE (Long integer)
- DATA (Blob)
- GLOBALID (GlobalID)

1 AN EXAMPLE

With arcpy:

```
import arcpy
import os

# Iterate over bus stops where the
ShelterType value is “covered” (2).
with arcpy.da.SearchCursor(
        r'C:\Transit.gdb\BusStop',
        where_clause='ShelterType = 2',
        field_names=['GlobalID', 'StopID']
        ) as cursor:

    # Map GlobalID to StopID.
    globalid_map = dict(list(cursor))
```

1 AN EXAMPLE

With arcpy:

```
# Construct the WHERE clause for retrieving
# attachment rows related to the bus stops
# returned above.
globalid_strings = [“’%s’” % (gid,) for gid
                    in globalid_map.keys()]

shelter_where = ‘REL_GLOBALID IN (%s)’ % (
  ‘, ‘.join(globalid_strings),)
```

1 AN EXAMPLE

With arcpy:

```
# Iterate over the selected attachments.
with arcpy.da.SearchCursor(
        r'C:\Transit.gdb\BusStop__ATTACH',
        where_clause=shelter_where,
        field_names=['REL_GLOBALID', 'DATA']
        ) as cursor:

    for row in cursor:
        # Unpack values from the row.
        rel_globalid, data = row
        # Find the stop ID.
        stop_id = globalid_map[rel_globalid]
```

1 AN EXAMPLE

With arcpy:

```
            # Construct the path to the photo.
            photo_path = r'C:\Photos\%i.jpg' % (
                stop_id,)
            # Save the photo if a file with that
            # path does not already exist.
            if not os.path.exists(photo_path):
                open(photo_path,
                     'wb').write(data.tobytes())
```

1 AN EXAMPLE

With cuuats.datamodel:

```
from cuuats.datamodel import D, \
    feature_class_factory as factory

# Create a Python class representing bus
# stops using introspection.
BusStop = factory(r'C:\Transit.gdb\BusStop')
```

1 AN EXAMPLE

With cuuats.datamodel:

```
# Iterate over bus stops where the
ShelterType value is “covered” (2).
for bus_stop in BusStop.objects.filter(
        ShelterType=D(‘Covered’)):

    # Select the first attachment, and save
    # it to the designated location.
    photo = bus_stop.attachments.first()
    if photo is not None:
        photo.save_to(
            r’C:\Photos’,
            ‘%i.jpg’ % (bus_stop.StopID,))
```

1 AN EXAMPLE

Differences with cuuats.datamodel:

- No need to deal with cursors or write SQL
- Interact with features as Python objects, not as lists
- Intuitive handling of relationships
- Shorter, easier-to-read code

2 WHY USE A DATA MODEL?

- Focus on what matters: data
- Write Python, not SQL
- Don't repeat yourself (DRY)
- Create reusable, easy-to-maintain code
- Use Python's object-oriented capabilities

2 WHY USE A DATA MODEL?

Inspired by Django:

- Popular Python web framework
- Includes a powerful object-relational mapper
- cuuats.datamodel uses a similar API:
  - Model ~ FeatureClass
  - QuerySet ~ QuerySet
- Excellent documentation: https://www.djangoproject.com/

3 CASE STUDY

- Inventory of public sidewalks, curb ramps, crosswalks and pedestrian signals in the Champaign-Urbana Urbanized Area
- Data collected with ArcGIS Collector on Android tablets:
  - 34,635 sidewalk features
  - 14,864 curb ramp features
  - 1,199 crosswalk features
  - 602 pedestrian signal features
- Goal: assess condition and ADA compliance
- Data model used for quality assurance, data maintenance, and analysis

4 BUILDING A PYTHON DATA MODEL

- Feature class factory
  - Auto-generates a Python class via introspection
  - Can follow relationships
- Explicit class definition
  - Allows more control over fields and model behavior
  - Documents data schema in code
  - Can extend (subclass) a factory-generated Python class

4 BUILDING A PYTHON DATA MODEL

Explicitly defining the Crosswalk model:

```
from cuuats.datamodel import FeatureClass, \
    NumericField, StringField

class Crosswalk(FeatureClass):
    SurfaceType = NumericField(
        'Surface Type',
        required=True)
Width = NumericField(
    'Width',
    required=True)
Comment = StringField(
    'Comment')
```

4 BUILDING A PYTHON DATA MODEL

Registering the Crosswalk model:

The Python class must be registered with the source feature class to enable data access. The attachment relationship is created during the registration process:

```
Crosswalk.register(
        r'C:\SidewalkInventory.gdb\Crosswalk')
```

5 QUERYING THE MODEL

Creating QuerySets:

As in Django, queries are executed using QuerySet objects. QuerySets are accessible via the “objects” attribute and represent a SQL query for selecting features:

```
>>> from cuuats.datamodel import D
>>> Crosswalk.objects.filter(
...     SurfaceType=D('Asphalt'))
<cuuats.datamodel.query.QuerySet object at
0x159372B0>
```

5 QUERYING THE MODEL

Evaluating QuerySets:

QuerySets are “lazy” and are not evaluated until we iterate over them:

```
>>> [cw for cw in Crosswalk.objects.filter(
...  SurfaceType=D('Asphalt'))]
[<Crosswalk: Crosswalk>, <Crosswalk:
Crosswalk>, ...]
```

5 QUERYING THE MODEL

Building complex QuerySets:

Filter calls can be chained to build more complex queries:

```
>>> cw = Crosswalk.objects.filter(
...       SurfaceType=D('Asphalt'))
>>> cw.filter(Width__gte=60).query.where
'SurfaceType = 2 AND Width >= 60'
```

Modifying field values:

We can retrieve a particular feature and access or modify its attributes:

```
>>> crosswalk = Crosswalk.objects.first()
>>> crosswalk.Width
72
>>> crosswalk.Width = 70
>>> crosswalk.SurfaceType = D('Concrete')
>>> crosswalk.save()
True
```

6 VALIDATING DATA

Performing validation:

Data models have a basic validation framework built-in:

```
>>> crosswalk = Crosswalk.objects.first()
>>> crosswalk.validate()
[]
```

Validation messages:

The feature class validate method calls the validate method for each field in the model and returns a list of validation error messages:

```
>>> crosswalk.Width = None
>>> crosswalk.validate()
['Width is missing']
```

6 VALIDATING DATA

Storing validation messages:

Validation error messages can be stored in a field:

```
for crosswalk in Crosswalk.objects.all():
    messages = crosswalk.validate()
    crosswalk.Comment = '; '.join(messages)
    crosswalk.save()
```

7 CALCULATING VALUES

The cuuats.datamodel package provides a framework for creating calculated fields that update automatically:

```
from cuuats.datamodel import ScaleField, \
    BreaksScale

class Crosswalk(FeatureClass):
    ...
    ScoreWidth = ScaleField(
        'Width Score',
        scale=BreaksScale(
            [36, 42, 48],
            [0, 33, 67, 100], False),
        value_field='Width')
```

Updating calculated fields:

Calculated fields are recalculated when they are retrieved:

```
>>> crosswalk = Crosswalk.objects.first()
>>> crosswalk.Width
72
>>> crosswalk.ScoreWidth
100
>>> crosswalk.Width = 36
>>> crosswalk.ScoreWidth
33
```

Saving calculated values:

Saving the feature will update calculated fields in the geodatabase:

```
>>> crosswalk.save()
True
```

Querying related features:

Adding a ForeignKey field to a feature class establishes a relationship between that class and the related class. Attachment relationships are added automatically:

```
>>> crosswalk = Crosswalk.objects.first()
>>> crosswalk.attachments.all()
<cuuats.datamodel.query.QuerySet object at 0x1559A7B0>
```

Traversing relationships backward:

Relationships are accessible from both sides:

```
>>> Attachment = Crosswalk.related_
classes['Attachment']
>>> attachment = Attachment.objects.first()
>>> attachment.feature
<Crosswalk: Crosswalk>
```

Querying across relationships:

Queries can span relationships by using the double underscores. Here we list only crosswalks where at least one of the attachments is larger than 1 MB:

```
>>> list(Crosswalk.objects.
filter(attachments__file_size__gt=1000000))
[<Crosswalk: Crosswalk>, <Crosswalk:
Crosswalk>, ...]
```

9 NEXT STEPS

- Feature creation and deletion
- Allow field values in queries
- Batch update (i.e., field calculator)
- Spatial relationships
- Documentation and more tests

Code: https://github.com/cuuats

Questions: Matthew J. Yoder (myoder@ccrpc.org)