

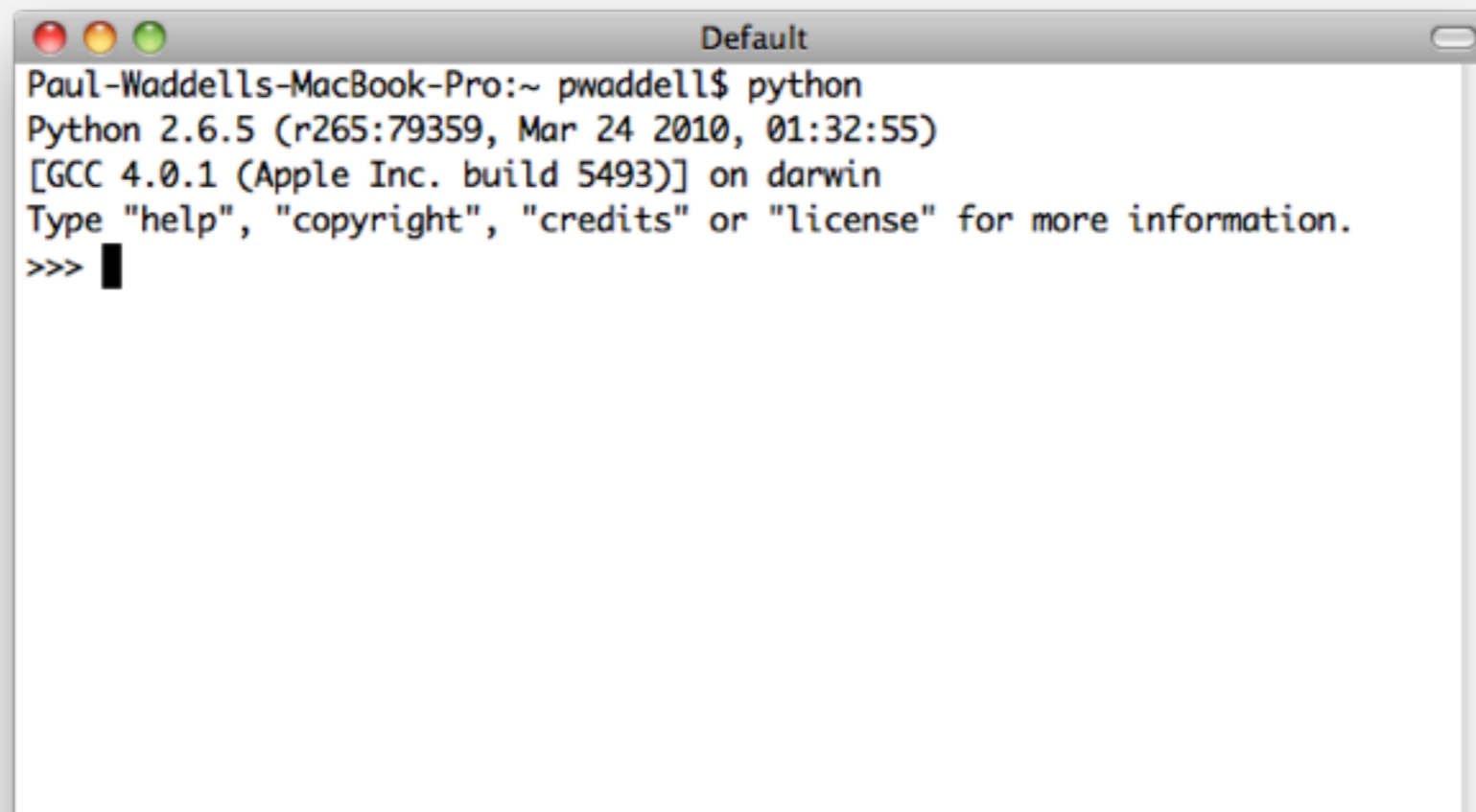


Introduction to Python, Numpy and the OPUS Expression Language

Paul Waddell
City and Regional Planning
University of California Berkeley

Starting Python

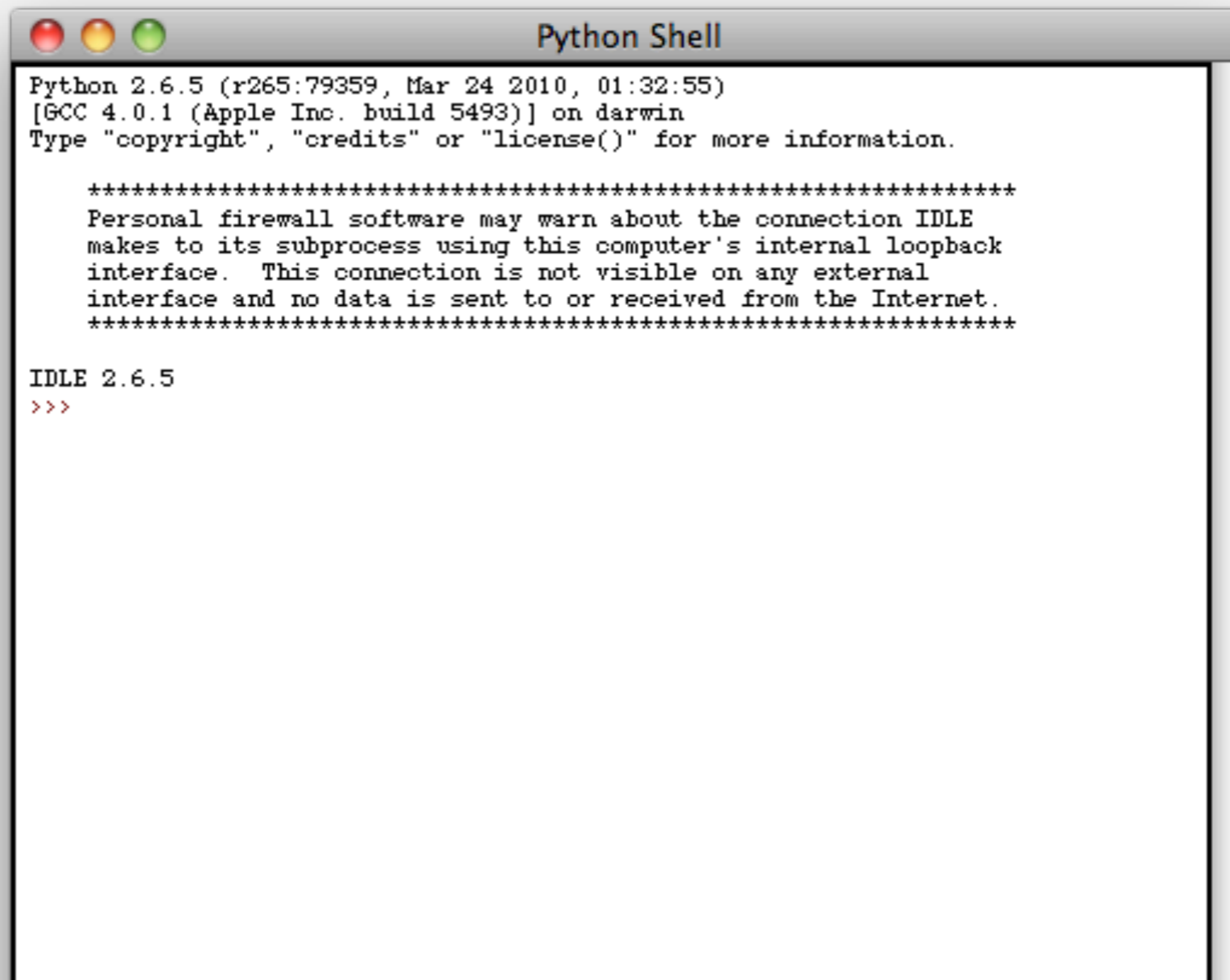
- There are two main ways of starting (launching) Python:
 - Starting an interactive session: for doing things ‘on the fly’, like using a calculator
 - Running a stored program: for running more complex tasks, like using a spreadsheet with many formulas
- The interactive session can be started from a command shell (in windows, this would involve start/run/cmd or finding the menu entry for the dos command window). On a mac, this would be the ‘terminal’ app (below), or in linux, a ‘shell’.



```
Paul-Waddells-MacBook-Pro:~ pwaddell$ python
Python 2.6.5 (r265:79359, Mar 24 2010, 01:32:55)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Starting Python

- Another way to start an interactive Python session is with IDLE, which is a simple GUI for Python that comes on all platforms. It is available from the Windows program menu, or can be started from the command line by typing 'idle'.



```
Python Shell
Python 2.6.5 (r265:79359, Mar 24 2010, 01:32:55)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface.  This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.5
>>>
```

Starting Python

- IDLE also has an editor, that allows you to write scripts, or programs, and save them to disk for running later. Note that Python programs on disk always have a .py filename suffix, like 'myprogram.py'. The .py indicates that it is Python.
- But let's stick with the interactive prompt for the moment. Notice that it has a prompt with three 'greater than' signs: >>>
- Here is your first program: make Python print 'Hello World!' (this is the first program in every programming language course)

- What is needed to do this?

Starting Python

- IDLE also has an editor, that allows you to write scripts, or programs, and save them to disk for running later. Note that Python programs on disk always have a .py filename suffix, like 'myprogram.py'. The .py indicates that it is Python.
- But let's stick with the interactive prompt for the moment. Notice that it has a prompt with three 'greater than' signs: >>>
- Here is your first program: make Python print 'Hello World!' (this is the first program in every programming language course)

- What is needed to do this?
- Pretty simple:

```
>>> print 'Hello World!'
```

```
Hello World!
```

Interactive Python: First Steps

- Let's begin by using Python for some interactive calculations. Note that we can simply type in a formula and Python will evaluate it:

```
>>> (2+3)/3.2
```

```
1.5625
```

- And we can assign values to variables and then operate on those variables:

```
>>> a = 2
```

```
>>> b = 3
```

```
>>> c = 3.2
```

```
>>> (a+b)/c
```

```
1.5625
```

Introducing Data Types

- Let's begin paying attention to data types. Numbers without decimals are Integers. Numbers with decimals are Floats. Let's see what happens if we use only integers with the preceding example:

```
>>> (2+3)/3
```

```
1
```

- Is this what you expected? Probably not. It is an integer division, and this results in a truncated result (no decimal value) with no rounding up.
- If you want to produce a result as a Float, at least one of the arguments (elements) of the calculation must be a float. That is why the previous example gave the 'right' answer.

Common Python Data Types and Simple Operations

- Integer: 1, 2, 3

```
>>> a = 1
```

- Float: 2.32592

```
>>> b = 3.141
```

- String: 'Abracadabra'

```
>>> c = 'Kalamazoo'
```

```
>>> c[0]      #(Python uses 'indexes to look things up -- the first entry is 0)
'K'
```

- List: ['apple', 'pear', 'orange'], [1, 5, 9, 12]

```
>>> d = ['apple', 'pear', 'orange']
```

```
>>> d[0:2]    #(Using a second index, you will get one less than you might expect)
['apple', 'pear']
```

Iteration and Methods

```
>>> def countdown(n):
...     while n > 0:
...         print n
...         n = n-1
...     print "Blastoff!"
...
>>> countdown(3)
3
2
1
Blastoff!
```

Note that the **def** keyword ‘defines’ a method in Python. It accepts an ‘argument’ in parentheses. It uses a colon to begin the definition part. Note also that the following lines are indented by four spaces. (this matters to Python - and is simpler than the syntax used in some other languages)

The ‘while’ statement defines an iterative loop, to continue indefinitely while the condition is true. To exit the loop, we have to change the value of *n* until the condition is no longer true. Note the indentation below this - also important.

Finally, note that to run the method, we call it by name, and pass an arbitrary argument to it, like ‘3’, or ‘10’ etc.

Python Modules

```
def countdown(n):  
    while n > 0:  
        print n  
        n = n-1  
    print "Blastoff!"  
countdown(n)
```

save this file to blastoff.py

Usually we would store longer commands or multi-line scripts like this in a Python module. If we use a text editor like the one built into IDLE, or use an alternative like Scite or TextMate (there are many options) that ‘understand’ Python syntax and highlight it and provide useful shortcuts, then save it to disk as a file with a name like `blastoff.py` (don’t call it `countdown.py` since the method has already used that name)

But we don’t want to ‘hard-code’ the number for `n` -- we want to pass it as an argument to the program. So we add 2 lines to the beginning of the program:

```
import sys  
n = int(sys.argv[1])
```

now we can run it from the command line, like this, passing the value of `n` as an argument:

```
c:\path_where_program_is_saved> python blastoff.py 10
```

Python Modules: Calling Methods Externally

```
def countdown(n):  
    while n > 0:  
        print n  
        n = n-1  
    print "Blastoff!"  
countdown(n)
```

save this file to blastoff.py

```
from blastoff import countdown  
  
countdown(10)
```

save this file to run_blastoff.py, or just run it interactively from the python prompt:

```
>>> from blastoff import countdown  
>>> countdown(10)
```

Reading and Writing Files

```
myfile = open('test.txt', 'w')
myfile.write('Some text\n')
myfile.write('goes here')
myfile.close()
```

*save this file to write_file.py
and then run it.*

```
myfile = open('test.txt', 'r')
print myfile.read()
myfile.close()
```

save this file to read_file.py, and run it:

```
python read_file.py
```

*Some text
goes here*

Reading and Writing Files

```
myfile = open('test.txt', 'w')  
myfile.write('Some text\n')  
myfile.write('goes here')  
myfile.close()
```

```
myfile = open('test.txt', 'r')  
print myfile.read()  
myfile.close()
```

*'w' or 'r' is the mode to open
the file with: write or read*

`\n` inserts a newline

Have to close a file after you
finish with it.

Introducing Numpy

- Numpy is a Python ‘library’ that provides numerical methods.
- It is somewhat similar in syntax to Matlab or R, in that it focuses on ‘arrays’
- It makes it possible to do repeated calculations fast by using an array rather than a for loop.
- Think of it as a spreadsheet, where a table of 3 rows and 4 columns is a 2-dimensional array
- Now generalize this to more than 2 dimensions and you have a multidimensional array. Numpy is meant to handle these arrays quickly.
- UrbanSim is implemented making heavy use of Numpy, so it is useful to learn its syntax before using the ‘expression language’ in UrbanSim and the Open Platform for Urban Simulation (OPUS)

Numpy Basics

- You have to import numpy to get its methods:

```
>>> import numpy
```

or

```
>>> from numpy import arange #or whatever list of methods you need
```

- Define an array with 10 elements from 0 to 9:

```
>>> a = numpy.arange(10)
```

```
>>> a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Now you can operate on this array all at once rather than iterating over it:

```
>>> a**2 #square each element
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Numpy Computations

- Now try dividing all the elements of a by 2:

```
>>> a/2
```

```
array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4])
```

- It did this as integer math -- so you get truncated results. Force it to a float to get the expected result. In general, keep track of data types in calculations.

```
>>> a/2.0
```

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
```

```
>>> a.dtype.name
```

```
'int32'
```

Common Numpy Methods

- Generate an array with 6 random numbers, shaped as 2 rows and 3 columns:

```
>>> a = random.random((2,3))
```

```
>>> a
```

```
array([[ 0.33135182,  0.60405674,  0.69395432],  
       [ 0.45638797,  0.54919324,  0.8084517 ]])
```

- find the sum, min and max of the array:

```
>>> print a.sum(), a.min(), a.max()
```

```
3.44339580179  0.331351823322  0.808451699071
```

- Reshape the array to a single row:

```
>>> a.reshape(6)
```

```
array([ 0.33135182,  0.60405674,  0.69395432,  0.45638797,  0.54919324,  
       0.8084517 ])
```

Common Numpy Methods

- Generate an array shaped as 2 rows and 3 columns, with each value as 2:

```
>>> b = ones((2,3))*2
```

```
>>> b
```

```
array([[ 2.,  2.,  2.],  
       [ 2.,  2.,  2.]])
```

- add a and b, element by element:

```
>>> a+b
```

```
array([[ 2.33135182,  2.60405674,  2.69395432],  
       [ 2.45638797,  2.54919324,  2.8084517  ]])
```

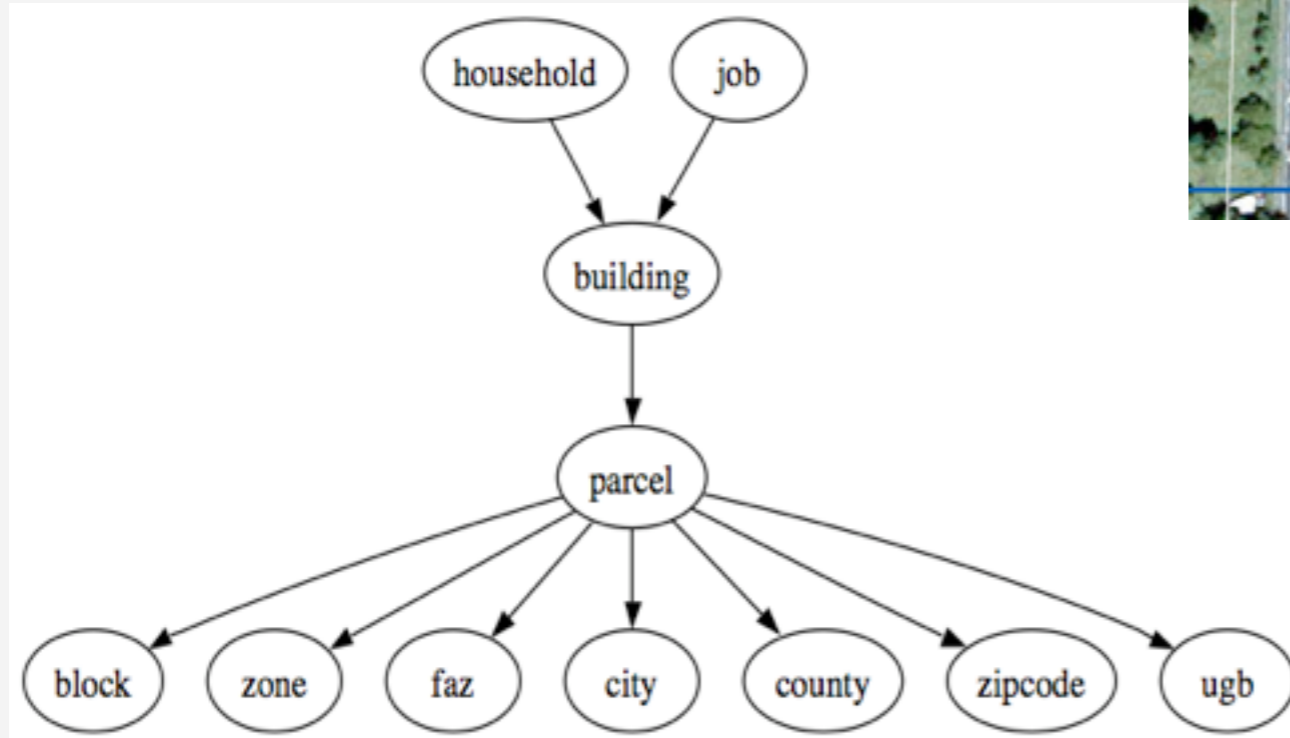
- Multiply a and b, element by element (note that this is not matrix multiplication):

```
>>> a*b
```

```
array([[ 0.66270365,  1.20811349,  1.38790865],  
       [ 0.91277594,  1.09838649,  1.6169034  ]])
```

UrbanSim Computations Require Spatial Referencing

- UrbanSim uses agents and objects that are related to each other and to locations. We have to keep track of these relationships and compute variables for models, and indicators for evaluation.



OPUS Datasets

- In OPUS, we create 'Datasets' that are collections of Numpy arrays describing all of the elements in a type of entity, like buildings, or parcels, or households
- The data are stored on disk as Numpy arrays, but are moved back and forth from databases using OPUS tools built into the GUI or from the command line
- There is a one-to-one correspondence between an OPUS Dataset and a corresponding database Table
- One (odd) convention we have used is that the name of a table in the database has been plural, while the counterpart OPUS Dataset is singular (this may be made consistent in the future, but was intended to reduce possible confusion)
- Datasets that are related to each other have some common key: the household dataset has a `building_id`, which corresponds to the internal id of the building dataset -- this allows complex joins and navigation among datasets in memory

OPUS Expression Language

- To make much of this easier for users to use and extend, we developed the ‘OPUS Expression Language’
- It is built on Python and Numpy, but adds syntax parsing that makes it possible to use more natural and concise commands to get complex work done
- Examples:
 - **parcel.aggregate(building.total_sqft)** #Sum the sqft in buildings in a parcel
 - **zone.aggregate(household.income, function=mean, intermediates=[building, parcel])** #Compute an average household income in a zone, navigating through the relationships that households are in buildings, which are in parcels, which are in zones
 - **parcel.aggregate(building.total_sqft)/parcel.lot_sqft** #Calculate a Floor Area Ratio (FAR)
 - **parcel.disaggregate(zone.population_density)** #this would assign to each parcel in a zone the value of the zone variable population_density
 - **zone.number_of_agents(job)** # Counts the number of jobs within a zone

OPUS Expression Language

- OPUS Expressions can use any of the Numpy standard operations:
 - **building.total_sqft/residential_units** #computes the average sqft per unit
 - **ln(building.total_sqft)** #takes the natural logarithm of total building sqft
 - Similarly, other **operators** include: `*`, `/`, `+`, `-`, `**`
 - results can be **cast** to a type: `ln(urbansim.gridcell.population).astype(float32)`
- References to variables and primary attributes
 - If an attribute is stored in a dataset and not calculated by OPUS ‘on the fly’, then we call it a **primary attribute**, and refer to it using the syntax **dataset.attribute** like `building.total_sqft`
 - If we want to refer to an attribute that is computed by a variable in OPUS, ‘on the fly’, then we have to make a reference to the Python module that computes it. These modules are stored within OPUS packages, which are directories. The syntax now becomes **package.dataset.variable_name**, like `urbansim_parcel.zone.access_to_employment`