Using Python for Analytics

“Batteries Included”

MarginHound
hound@marginhound.com
July 14th 2011
The Analyst Role (and its discontents)

• Theoretical Purpose of The Analyst – Guide other members of the team through developing rigorous solutions to business questions, using deep expertise in statistics, finance, etc.

• How it really works:
  – In retrospect, the answer to most “valuable” questions becomes obvious if you have access to the correct data
  – Corollary: If we had the data, we would have already solved it.
  – Your job is to figure out how to hack together the right dataset
    • From stuff we haven’t used before…
    • And make sure it’s right…
    • By noon tomorrow…
Why Python?

• Ad-hoc analysis usually requires three “layers” in your tool box:
  – Data Extraction => SQL or a query builder
  – Transformation & Analysis => Scripting Language
  – Presentation => Excel / PowerPoint / Access

• Python handles the middle layer well:
  – Succinct, Powerful Code – Duck Typing, First Class Functions
  – More expressive than databases (SQL), MS Office, statistics applications
  – Large library of built-in modules / data-types for common chores
  – Easy access to higher speed options (Numpy, Cython, JIT compilers)
  – Interpreter – often have to “doodle” with data / functions to identify trends
  – Readability Counts…
Hypothetical Problem (For Main Examples)

- You manage a kitchen
- Every day, you purchase food – you keep a record of this
- You would like to know:
  - What you are buying?
  - How much variation in prices exists?
- You intend to ask for a discount but…
  - Your customers are very sensitive to certain items – so you need to make sure you don’t lose those suppliers…
- Naturally, as a trained analyst, you want to use statistics which aren’t easily available in most entry-level database programs…
Typical Process Flow

Databases
Oracle & Access

Manipulate Data
(Merge / Transform)

Core Python
- List Comprehensions
- Itertools

Numpy
- Matplotlib

Statistical Calculations
- Numpy Array
- Scipy Library

Presentation Layer

Excel & Flat Files

CSV

Pyodbc
CeODBC

Typical Process Flow
Python ODBC Connections

• Simplifies Access to Databases
  – Script refresh / download processes (eliminates boring work)
  – Save “snapshots” of data sets for future review / justification
  – Can directly export your results to DB/Access/Excel

• Requires you to learn SQL but..
  – Enables you to shift some of your processing to the larger machine; increases calculation speed and reduces volume of data to retrieve
  – High end databases will often have a nice analytics library

• Python has a database API specification (v 2.0); many libraries exist (mix of free and commercial). Two good ones:
  – pyodbc http://code.google.com/p/pyodbc/ (supports 2.x)
  – ceODBC http://ceodbc.sourceforge.net/ (supports 3.1)
Python ODBC Connections

High Level Activities

- Establish a connection to the database
- Set up a cursor
- Generate SQL & feed it to the cursor
- Cursor returns an iterable object for you to work with
- If writing, remember to commit
- Close connection when done – risks locking the database / table

Read Example

```python
conn = pyodbc.connect('DSN=Main DW';
    UID=SPAM;PWD=EGGS)
cursor = conn.cursor()
sql = "Select food, amount from po_list"
cursor.execute (sql)
Results = [ [row.food, row. amount]) for row in cursor]
```

Write Example

```python
<continues read example>
sql="Insert into po_list (food, amount)
    values (‘Milk’, 1)"
cursor.execute(sql)
conn.commit()
conn.close()
```
Databases – Simplifying your life

• Analytics SQL is often verbose, repetitive, and tedious to debug:
  • 80% of your queries request the same data (invoices, shipments, etc)
  • A solution – build templates, customize at runtime (search/replace)
  • Use external template files so you can share within your team

• Multi-step queries are frequently simpler / faster / more transparent:
  • Run an initial query to get current database status (update dates, etc.)
  • Complete calculation of query parameters in your script
  • Update the main query template(s) with the results of your work
  • Often much easier to test, may execute quicker as well

• Also worth automating “data type conversion” when using the results:
  • Cursor object has a “description” attribute – data type, size, etc.
  • Write introspective code to identify data types (for array, table creation)
  • Also useful for: date conversions, management of null values
Manipulating Data – Core Python

Can get a lot done using simple fundamentals:

• Data Structures
  • Obvious Choices - List of Lists (aka Nested Lists), Dictionaries
  • Specialized Options: Deque, Array, Tuples, Named Tuples

• Useful tools for slicing and dicing
  • List comprehensions
    • Select / Filter data, apply functions to results
    • Use nested and multi-step list comps for advanced operations
    • enumerate() – useful for ranking, updating a nested list in place
  • If-Else Expressions (X if X>0 Else 0)
  • Lambda and Map
  • Operator.Itemgetter() - allows you to select subset of elements from a list
  • Itertools: Group By, Cycles, Calculate Permutations

• Constantly making tradeoffs within this universe:
  • Remembering Data Structure Layout (Field1, Field2, Field3)
  • Additional complexity of manipulating data structures other than lists / tuples
  • Processing Performance impacts
Manipulating Data – List of Lists Examples

We will start manipulating the purchase order data for our main example:

Each record consists of:

1 – food
2 – uom
3 – amount
4 - unit_cost
5 - buy_date

• Example 1 – calculate total cost for each PO (item 2 x item 3)

```
    dataset= [item + [item[2]*item[3]]
              for item in dataset]
```

• Example 2 – rank my purchase orders by total cost

```
    dataset = [[i+1] + item for i, item in
               enumerate(
               sorted(dataset, key=operator.itemgetter(5),
                        reverse = True))]
```
Manipulating Data – More Complex

Want to replicate functionality delivered by a SQL “Group By” Statement and aggregate statistical calculations, with the following twists:

• Define your own aggregate statistics (Python, Numpy, custom code)
• Incorporate data from outside your original database
• Wants to be able to recycle code within your script (similar reports)

The Specific Request (using our kitchen example):

• Group purchase orders by item purchased (spam, eggs, beer, etc.)
• Check to see if there are special notes for the item
• Calculate list of aggregate statistics (total qty, total cost, best cost, etc.)

Solution Components (using some “helper” functions):

• Group records using itertools.groupby
• Use dictionary “get” method to append notes to the keys
• Use list comprehension to calculate statistics for each group
Manipulating Data – More Complex

First Helper - set up the group by statement

The Function:

```python
def group_my_list(dataset, my_key):
    return itertools.groupby(sorted(dataset, key=my_key),
    key=my_key)
```

The Function Call:

```python
    group_my_list(dataset, operator.itemgetter(0))
```

Explanation:

- Returns a “configured” group by iterator with less repetitive code
- Dataset needs to be sorted by your key
- Key is actually a comparison function
  - Use operator.itemgetter to select specific list element
  - Could rewrite this to pass the list element vs. a function
Manipulating Data – More Complex

Second Helper – Process List of Statistics For Each Group

The Function:

```python
def run_stats(record_set, stats_list):
    return [stat(map(operator.itemgetter(ref), record_set))
            for stat, ref in stats_list]
```

The Function Call:

```python
PO_Stats = ((sum,2),(sum,5),(min,2),(max,2),(min,3),(max,3), (np.median, 3))
run_stats(list(g),PO_Stats)
```

Explanation:

- Called with a list of grouped records and a list of function / element pairs
- Returns a list of aggregate statistics (one per pair on function list)
- For each pair in the list of the function / element pairs:
  - Use map and operator.itemgetter to select a list of that element (eg. all prices)
  - Use the function piece of the pair to reduce that list to a single value
  - Append that value to your result list
Manipulating Data – More Complex

Bringing It all Together…. (doing the dictionary check in-line)

Generating The List of Aggregate Statistics:

\[
\text{prod_agg} = \left[ [k + \text{special_prefs.get}(k,\text{""})] + \right.
\]
\[
\left. \quad \text{run_stats}(\text{list}(g),\text{PO_Stats}) \right.
\]
\[
\quad \text{for } k, g \text{ in }\]
\[
\left. \quad \text{group_my_list}(	ext{dataset, operator.itemgetter(0)}) \right]\]

Explanation:

- Use a list comprehension to iterate across the sets of grouped records
- For each set of grouped records, construct a “result list” by joining:
  - The key plus any notes (Single Element List)
    - For each key, use the get method of the dictionary “special prefs” to return any notes; get lets you define a default value (in this case, ””)
  - The aggregate statistics for that group
    - Calculated using our run_stats function
Manipulating Data – Demonstration

• Generated table of product level statistics using our group by statement
• Calculated a potential savings number (cost @ best price vs. actual cost)
• Ranked categories by potential savings
• And generated the following graph (using matplotlib.pyplot’s plot function):

Looks like we need to visit the cheese shop....
Manipulating Data – Numpy

Numpy Array:

- Wraps a very fast low level array for performing calculations
- Supported by set of built-in function optimized for numpy
  - Some of these functions can also be use on Python lists
- Large base of supporting statistical/numeric libraries in Scipy
- The price: must define data type in advance, one type per array
- But…may significant boost performance (with minor changes)

  Analyzing 5MM subsets of series: 600 -> 150 CPU seconds

Structured Array:

- Variant of the numpy array but can access “columns” of data using string references (eg. “price”, “cost”, “part number”)
- Similar to data sets / frames in R, SAS, and other languages
- Significantly More Readable – certain operations may be slower
- Can mix data types (at the column level)
- Very good for exploratory analysis
Manipulating Data – Matplotlib

• Plotting Library for Scipy / Numpy
  • Mlab module has utilities for managing datasets / structured arrays

• Some useful functions
  • Rec_summarize Create New Field by Applying a Function
  • Rec_GroupBy Aggregate Stats for Subset of Records
  • Rec_Append_Fields Create New Field from like-sized array

• Other useful functions
  • Rec_join Match datasets
  • Drop_fields Simplify datasets
  • Rec2CSV, CSV2REC Load / Unload datasets (auto-typing)
Manipulating Data – Numpy Example

Some sample applications:

**Example 1 – Calculating a field using other fields**

```python
dataset = ml.rec_append_fields(dataset, "gross_cost", [item['amount'] * item['unit_cost'] for item in dataset])
```

**Example 2a - Create New Field using dict lookup**

```python
lookup = (('food', lambda x: [item + special_prefs.get(item, "") for item in x], 'food_groups'),)
prod_agg = ml.rec_summarize (prod_agg, lookup)
```
Manipulating Data – Numpy Example

Example 2b – Group By With Aggregate Statistics

```python
stats_list = (("amount",np.sum,"qty_sum"),
              ("gross_cost",np.sum,"cost_sum"),
              ("amount",np.min,"qty_min"),
              ("amount",np.max,"qty_max"),
              ("unit_cost",np.min,"uc_min"),
              ("unit_cost",np.max,"uc_max"),
              ("unit_cost",np.median,"uc_median"),)

prod_agg = ml.rec_groupby(dataset,(("food"),),stats_list)
```

“The Hack”

These functions appear to work with any function which:

- Rec_Summarize accepts a list and returns a list of the same size/order
- Rec_GroupBy accepts a list and reduces it to a single value

Which enables you to execute a wide range of calculations and transformations with some creative use of list comprehensions and other methods.
Summation

• Several ways to do it – the “obvious one” depends on tradeoffs
  – How much does your data structure change?
  – Is the data fundamentally static (eg. financial markets data)
  – Developer Speed vs. Processing Power

• Don’t underestimate the value of freedom
  – Create / Extend your own analytical functions
  – Develop your own frameworks / helper libraries
  – Can view / fork source code for key modules
  – Active online support community

• Makes the analyst role more interesting
  – Can ask questions faster, streamline repetitive tasks
  – Transparency -> Quality -> Less Stress
  – More time to think of interesting ways to transform your data