Reading Multidimensional Containers

Python 2

home (../handouts.html)

Introduction: Reading Multidimensional Containers

Data can be expressed in complex ways using *nested* containers.

Real-world data is often more complex in structure than a simple sequence (i.e., a list) or a collection of pairs (i.e. a dictionary).

- A student database of unique student ids, with several address and billing fields associated with each id
- A listing of businesses with market cap, revenue, etc. associated with each business
- A list of devices on a network, each with attributes associated with each (id, firmware version, uptime, latency)
- A log listing of events on a web server, with attributes of the event (time, requesting ip, response code) for each
- A more complex nested structure as might be expressed in an XML or HTML file Complex data can be structured in Python through the use of *multidimensional containers*, which are simply containers that contain other containers (lists of lists, lists of dicts, dict of dicts, etc.) in structures of *arbitrary complexity*. Most of the time we are not called upon to handle structures of greater than 2 dimensions (lists of lists, etc.) although some config and data transmitted between systems (such as API responses) can go deeper. In this unit we'll look at the standard 2-dimensional containers we are more likely to encounter or want to build in our programs.

Objectives for the Unit: Reading Multidimensional Containers

- Identify and visually/mentally parse 2-dimensional structures: *lists of lists*, *lists of dicts*, *dicts of dicts* and *dicts of lists*
- Read any value within a multidimensional structure through chained subscripts
- Loop through and print selected values within a multidimensional structure

Summary Structure: List of Lists

A list of lists provides a "matrix" structure similar to an Excel spreadsheet.

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]
```

Probably used more infrequently, a list of lists allows us to access values through list methods (looping and indexed subscripts).

The "outer" list has 3 items -- each items is a list, and each list represents a row of data.

Each row list has 4 items, which represent the row data from the Fama-French file: the date, the Mkt-RF, SMB, HML and RF values.

Looping through this structure would be very similar to looping through a delimited file, which after all is an iteration of lines that can be split into fields.

```
for rowlist in value_table:
    print "the MktRF for {} is {}".format(rowlist[0], rowlist[1])
```

Summary Structure: List of Dicts

A list of dicts structures tabular rows into field-keyed dictionaries.

```
value_table = [
                { 'date': '19260701', 'MktRF': 0.09, 'SMB': -0.22, 'HML':
                { 'date': '19260702', 'MktRF': 0.44, 'SMB': -0.35, 'HML':
                { 'date': '19260706', 'MktRF': 0.17, 'SMB': 0.26,  'HML':
              ]
```

The "outer" list contains 3 items, each being a dictionary with identical keys. The keys in each dict correspond to field / column labels from the table, so it's easy to identify and access a given value within a row dict.

A structure like this might look elaborate, but is very easy to build from a data source. The convenience of named subscripts (as contrasted with the numbered subscripts of a list of lists) lets us loop through each row and name the fields we wish to access:

```
for rowdict in value_table:
    print "the MktRF for {} is {}".format(rowdict['date'], rowdict['MktRF'
```

Summary Structure: Dict of Lists

A dict of lists allows association of a sequence of values with unique keys.

```
value_table = { '1926': [  0.09,  0.44,  0.17, -0.15, -0.06, -0.55,  0.61,
                '1927': [ -0.97,  0.30,  0.13, -0.18,  0.31,  0.39,  0.14,
                '1928': [  0.43, -0.14, -0.71,  0.61,  0.13, -0.88, -0.85,
```

The "outer" dict contains 3 string keys, each associated with a list of float values -- in this case, the MktRF values from each of the trading days for each year (only the first 9 are included here for clarity).

With a structure like this, we can perform calculations like those we have done on this data for a given year, namely to identify the **max()**, **min()**, **sum()**, average, etc. for a given year

```
for year in value_table:
    print 'for year {}:  len {}, sum {}, avg {}'.format(year,
                                                       len(value_table[ye
                                                       sum(value_table[ye
                                                       (sum(value_table[y
```

Summary Structure: Dict of Dicts

In a dict of dicts, each unique key points to another dict with keys and

values.

```
date_values = {
  '19260701':   { 'MktRF':  0.09,
                  'SMB':   -0.22,
                  'HML':   -0.30,
                  'RF':    0.009 },
  '19260702':   { 'MktRF':  0.44,
                  'SMB':   -0.35,
                  'HML':   -0.08,
                  'RF':    0.009 },
}
```

The "outer" dict contains string keys, each of which is associated with a dictionary -- each "inner" dictionary is a convenient key/value access to the fields of the table, as we had with a list of dicts.

Again, this structure may seem complex (perhaps even needlessly so?). However, a structure like this is extremely easy to build and is then very convenient to query. For example, the 'HML' value for July 2, 1926 is accessed in a very visual way:

```
print date_values['19260702']['HML']        # -0.08
```

Summary Structure: arbitrary dimensions

Containers can nest in "irregular" configurations, to accomodate more complex orderings of data.

See if you can identify the object type and elements of each of the containers represented below:

```
conf = [
  {
    "domain": "www.example1.com",
    "database": {
      "host": "localhost1",
      "port": 27017
    },
    "plugins": [
       "plugin1",
       "eslint-plugin-plugin1",
       "plugin2",
       "plugin3"
    ]
  },   # (additional dicts would follow this one in the list)
]
```

Above we have a list with one item! The item is a dictionary with 3 keys. The "domain" key is associated with a string value. The "database" key is associated with another dictionary of string keys and values. The "plugins" key is associated with a list of strings.

Presumably this "outer" list of dicts would have more than one item, and would be followed by additional dictionaries with the same keys and structure as this one.

Summary Task: retrieving an "inner" element value

Nested subscripts are the usual way to travel "into" a nested structure to obtain a value.

A list of lists

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]

print "SMB for 7/3/26 is {}".format(value_table[2][2])
```

A dict of dicts

```
date_values = {
  '19260701':   { 'MktRF':  0.09,
                  'SMB':   -0.22,
                  'HML':   -0.30,
                  'RF':    0.009 },
  '19260702':   { 'MktRF':  0.44,
                  'SMB':   -0.35,
                  'HML':   -0.08,
                  'RF':    0.009 },
}

MktRF_thisday = date_values['19260701']['MktRF']   # value is 0.09

print date_values['19260701']['SMB']               # -0.22
print date_values['19260701']['HML']               # -0.3
```

Summary Task: looping through a complex structure

Looping through a nested structure often requires an "inner" loop within an "outer" loop.

looping through a list of lists

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]

for row in value_table:
    print "MktRF for {} is {}".format(row[0], row[1])
```

looping through a dict of dicts

```
date_values = {
  '19260701':   { 'MktRF':  0.09,
                  'SMB':   -0.22,
                  'HML':   -0.30,
                  'RF':    0.009 },
  '19260702':   { 'MktRF':  0.44,
                  'SMB':   -0.35,
                  'HML':   -0.08,
                  'RF':    0.009 },
}

for this_date in date_values:
    print "MktRF for {} is {}".format(this_date, date_values[this_date]['M
```

Multidimensional structures - building

Usually, we don't initialize multi-dimensional structures within our code. Sometimes one will come to us, as with **dict.items()**, which returns a list of tuples. Database results also come as a list of tuples.

Most commonly, we will build a multi-dimensional structure of our own design based on the data we are trying to store. For example, we may use the Fama-French file to build a dictionary of lists - the key of the dictionary being the date, and the value being a 4-element list of the values for that date.

```
outer_dict = {}                                          # new dict
for line in open('F-F.txt').read().splitlines()[4:-3]:
  columns = line.split()                                 # split each line
  date = columns[0]                                      # the first value
  values = columns[1:]                                   # slice this list
  outer_dict[date] = values                              # so values is a
                                                         # thus we have bu
```

Perhaps we want to be more selective - build the inner list inside the loop:

```
outer_dict = {}                                         # new dict
for line in open('F-F.txt').read().splitlines()[4:-3]:
  inner_list = []                                        # a new, empty 'i
  columns = line.split()                                 # split the line
  date = columns[0]                                      # the first value
  inner_list.append(columns[1])                          # add the 1st num
  inner_list.append(columns[4])                          # add the 4th num
  outer_dict[date] = inner_list                          # now assign this
```

Inside the loop we are creating a temporary, empty list; building it up; and then finally associating it with a key in the "outer" dictionary. The work inside the loop can be seen as "the life of an inner structure" - it is built and then added to the inner structure - and then the loop moves ahead one line in the file and does the work again with a new inner structure.

Summary Function: pprint

pprint() prints a complex structure in readable format.

```
import pprint

dvs = {'19260701': {'HML': -0.3, 'RF': 0.009, 'MktRF': 0.09, 'SMB': -0.22}

pprint.pprint(dvs)

       ### {'19260701': {'HML': -0.3, 'MktRF': 0.09, 'RF': 0.009, 'SMB': -
       ###  '19260702': {'HML': -0.08, 'MktRF': 0.44, 'RF': 0.009, 'SMB':
```