DSC 201: Data Analysis & Visualization

Data Frames

Dr. David Koop
Iterators

- Remember `range, values, keys, items`?
- They return **iterators**: objects that traverse containers, only need to know how to get the next element
- Given iterator `it`, `next(it)` gives the next element
- **StopIteration** exception if there isn't another element
- Generally, we don't worry about this as the for loop handles everything automatically
- ...but you cannot index or slice an iterator
- `d.values()[0]` and `range(100)[-1]` will not work!
- If you need to index or slice, construct a list from an iterator
- `list(d.values())[0]` or `list(range(100))[-1]`
- In general, this is slower code so we try to avoid creating lists
List and Dictionary Comprehensions

• List Creation
  • squares = []
    for i in range(10):
      squares.append(i**2)
  • squares = [i**2 for i in range(10)]

• Filtering (if after for)
  • squares = [i**2 for i in range(10) if i % 3 != 1]

• Dictionary Comprehension
  • squares = {}
    for i in range(10):
      squares[i] = i**2
  • squares = {i: i**2 for i in range(10)}
Exceptions

• errors but potentially something that can be addressed
• try-except: except allows specifications of exactly the error(s) you wish to address

• finally: always runs (even if the program is about to crash)
• can also raise exceptions using the raise keyword
• …and define your own
Assignment 2

- http://www.cis.umassd.edu/~dkoop/dsc201/assignment2.html
- Analyze data on refugees
- Requires cleaning up the data
- Data stored in dictionaries
Arrays

- Usually a fixed size—lists are meant to change size
- Are mutable—tuples are not
- Store only one type of data—lists and tuples can store anything
- Are faster to access and manipulate than lists or tuples
- Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.
Creating arrays

- `data1 = [6, 7.5, 8, 0, 1]`
  `arr1 = np.array(data1)`
- `data2 = [[1,2,3,4],[5,6,7,8]]`
  `arr2 = np.array(data2)`
- Number of dimensions: `arr2.ndim`
- Shape: `arr2.shape`
- Types: `arr1.dtype`, `arr2.dtype`, can specify explicitly (`np.float64`)
- Zeros: `np.zeros(10)`
- Ones: `np.ones((4,5))`
- Empty: `np.empty((2,2))`
- _like versions: pass an existing array and matches shape with specified contents
- Range: `np.arange(15)`
Array Slicing

Expression | Shape
---|---

In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)

In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792],
       [ 2.1452,  0.8799]])

[W. McKinney, Python for Data Analysis]
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

\[
\text{In [87]: names == 'Bob'}
\]

\[
\text{Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)}
\]

This boolean array can be passed when indexing the array:

\[
\text{In [88]: data[names == 'Bob']}
\]

\[
\text{Out[88]: array([[-0.048 ,  0.5433, -0.2349,  1.2792],}
\]

\[
[ 2.1452,  0.8799, -0.0523,  0.0672]]
\]

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

\[
\text{In [89]: data[names == 'Bob', 2:]}\]

\[
\text{Out[89]: array([[-0.2349,  1.2792],}
\]

\[
86 | Chapter 4: NumPy Basics: Arrays and Vectorized Computation

Array Slicing

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[:, 1:]</td>
<td>(2, 2)</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Array Slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as \(==\)) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

[W. McKinney, Python for Data Analysis]
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array(
    [[-0.048 ,  0.5433, -0.2349,  1.2792],
     [ 2.1452,  0.8799, -0.0523,  0.0672]],
    dtype=float64)
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array(
    [[-0.2349,  1.2792]],
    dtype=float64)
```

---

Array Slicing

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arr[:2, 1:]</code></td>
<td><code>(2, 2)</code></td>
</tr>
<tr>
<td><code>arr[2]</code></td>
<td><code>(3,)</code></td>
</tr>
<tr>
<td><code>arr[2, :]</code></td>
<td><code>(3,)</code></td>
</tr>
<tr>
<td><code>arr[2::]</code></td>
<td><code>(1, 3)</code></td>
</tr>
<tr>
<td><code>arr[:, :2]</code></td>
<td><code>(3, 2)</code></td>
</tr>
</tbody>
</table>
Array Slicing

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[:, 1:]</td>
<td>(2, 2)</td>
</tr>
<tr>
<td>arr[2]</td>
<td>(3, 1)</td>
</tr>
<tr>
<td>arr[2, :]</td>
<td>(3, 1)</td>
</tr>
<tr>
<td>arr[2:, :]</td>
<td>(1, 3)</td>
</tr>
<tr>
<td>arr[:, 2]</td>
<td>(3, 2)</td>
</tr>
<tr>
<td>arr[1, :2]</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>arr[1:2, :2]</td>
<td>(1, 2)</td>
</tr>
</tbody>
</table>

Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)

This boolean array can be passed when indexing the array:

In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])

[W. McKinney, Python for Data Analysis]
Boolean Indexing

• names == 'Bob' gives back booleans that represent the element-wise comparison with the array names

• Boolean arrays can be used to index into another array:
  - data[names == 'Bob']

• Can even mix and match with integer slicing

• Can do boolean operations (&, |) between arrays (just like addition, subtraction)
  - data[(names == 'Bob') | (names == 'Will')]

• Note: or and and do not work with arrays

• We can set values too!
  - data[data < 0] = 0
Other Operations

• Fancy Indexing: \( \text{arr}[[1, 2, 3]] \)

• Transposing arrays: \( \text{arr}.T \)

• Reshaping arrays: \( \text{arr}.\text{reshape}((3, 5)) \)

• Unary universal functions (ufuncs): \( \text{np}.\text{sqrt}, \text{np}.\text{exp} \)

• Binary universal functions: \( \text{np}.\text{add}, \text{np}.\text{maximum} \)
## Unary ufuncs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs, fabs</td>
<td>Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data</td>
</tr>
<tr>
<td>sqrt</td>
<td>Compute the square root of each element. Equivalent to ( \text{arr} ** 0.5 )</td>
</tr>
<tr>
<td>square</td>
<td>Compute the square of each element. Equivalent to ( \text{arr} ** 2 )</td>
</tr>
<tr>
<td>exp</td>
<td>Compute the exponent ( e^x ) of each element</td>
</tr>
<tr>
<td>log, log10, log2, log1p</td>
<td>Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively</td>
</tr>
<tr>
<td>sign</td>
<td>Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)</td>
</tr>
<tr>
<td>ceil</td>
<td>Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element</td>
</tr>
<tr>
<td>floor</td>
<td>Compute the floor of each element, i.e. the largest integer less than or equal to each element</td>
</tr>
<tr>
<td>rint</td>
<td>Round elements to the nearest integer, preserving the dtype</td>
</tr>
<tr>
<td>modf</td>
<td>Return fractional and integral parts of array as separate array</td>
</tr>
<tr>
<td>isnan</td>
<td>Return boolean array indicating whether each value is NaN (Not a Number)</td>
</tr>
<tr>
<td>isfinite, isinf</td>
<td>Return boolean array indicating whether each element is finite (non-infinity, non-NaN) or infinite, respectively</td>
</tr>
<tr>
<td>cos, sinh, tan, tanh</td>
<td>Regular and hyperbolic trigonometric functions</td>
</tr>
<tr>
<td>arcsinh, arccosh, arctan, arctanh</td>
<td>Inverse trigonometric functions</td>
</tr>
</tbody>
</table>

### Table 4-4. Binary universal functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add corresponding elements in arrays</td>
</tr>
<tr>
<td>subtract</td>
<td>Subtract elements in second array from first array</td>
</tr>
<tr>
<td>multiply</td>
<td>Multiply array elements</td>
</tr>
<tr>
<td>divide, floor_divide</td>
<td>Divide or floor divide (truncating the remainder)</td>
</tr>
<tr>
<td>power</td>
<td>Raise elements in first array to powers indicated in second array</td>
</tr>
<tr>
<td>maximum, fmax</td>
<td>Element-wise maximum. ( \text{fmax} ) ignores NaN</td>
</tr>
<tr>
<td>minimum, fmin</td>
<td>Element-wise minimum. ( \text{fmin} ) ignores NaN</td>
</tr>
<tr>
<td>mod</td>
<td>Element-wise modulus (remainder of division)</td>
</tr>
<tr>
<td>copysign</td>
<td>Copy sign of values in second argument to values in first argument</td>
</tr>
</tbody>
</table>

[D. Koop, DSC 201, Fall 2016] [W. McKinney, Python for Data Analysis]
## Binary ufuncs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add corresponding elements in arrays</td>
</tr>
<tr>
<td>subtract</td>
<td>Subtract elements in second array from first array</td>
</tr>
<tr>
<td>multiply</td>
<td>Multiply array elements</td>
</tr>
<tr>
<td>divide, floor_divide</td>
<td>Divide or floor divide (truncating the remainder)</td>
</tr>
<tr>
<td>power</td>
<td>Raise elements in first array to powers indicated in second array</td>
</tr>
<tr>
<td>maximum, fmax</td>
<td>Element-wise maximum. fmax ignores NaN</td>
</tr>
<tr>
<td>minimum, fmin</td>
<td>Element-wise minimum. fmin ignores NaN</td>
</tr>
<tr>
<td>mod</td>
<td>Element-wise modulus (remainder of division)</td>
</tr>
<tr>
<td>copysign</td>
<td>Copy sign of values in second argument to values in first argument</td>
</tr>
</tbody>
</table>
| greater, greater_equal, less, less_equal, equal, not_equal | Perform element-wise comparison, yielding boolean array. Equivalent to infix operators \\
| logical_and, logical_or, logical_xor | Compute element-wise truth value of logical operation. Equivalent to infix operators & |

[W. McKinney, Python for Data Analysis]
# Statistical Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.</td>
</tr>
<tr>
<td>mean</td>
<td>Arithmetic mean. Zero-length arrays have NaN mean.</td>
</tr>
<tr>
<td>std, var</td>
<td>Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).</td>
</tr>
<tr>
<td>min, max</td>
<td>Minimum and maximum.</td>
</tr>
<tr>
<td>argmin, argmax</td>
<td>Indices of minimum and maximum elements, respectively.</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum of elements starting from 0</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product of elements starting from 1</td>
</tr>
</tbody>
</table>

### Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the above methods. Thus, `sum` is often used as a means of counting True values in a boolean array:

```
In [160]: arr = randn(100)
In [161]: (arr > 0).sum() # Number of positive values
Out[161]: 44
```

There are two additional methods, `any` and `all`, useful especially for boolean arrays.

- `any` tests whether one or more values in an array is True, while `all` checks if every value is True:

```
In [162]: bools = np.array([False, False, True, False])
In [163]: bools.any()
Out[163]: True
In [164]: bools.all()
Out[164]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

### Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place using the `sort` method:

```
In [165]: arr = randn(8)
In [166]: arr
Out[166]: array([ 0.6903,  0.4678,  0.0968, -0.1349,  0.9879,  0.0185, -1.3147, -0.5425])
In [167]: arr.sort()
```

---

[W. McKinney, Python for Data Analysis]
Other Functions

• *any* and *all*
• *sort*
• *unique*
• Other set operations
pandas

- Contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python
- Built on top of NumPy
- Requirements:
  - Data structures with labeled axes (aligning data)
  - Time series data
  - Arithmetic operations that include metadata (labels)
  - Handle missing data
  - Merge and relational operations
Pandas Code Conventions

- `from pandas import Series, DataFrame`
- `import pandas as pd`
Series

- A one-dimensional array with an **index**
- Index defaults to numbers but can also be text (like a dictionary)
- Allows easier reference to specific items
- Has an associated type just like a NumPy array
- `obj = Series([7,14,-2,1])`
- Basically two arrays: `obj.values` and `obj.index`
- Can specify the index explicitly and use strings
  - `obj2 = Series([4, 7, -5, 3],
                    index=['d', 'b', 'a', 'c'])`
- Could think of a fixed-length, ordered dictionary
- Can create from a dictionary
  - `obj3 = Series({'Ohio': 35000, 'Texas': 71000,
                      'Oregon': 16000, 'Utah': 5000})`
Series

• Indexing: s[1] or s['Oregon']
• Can check for missing data: `pd.isnull(s)` or `pd.notnull(s)`
• Both index and values can have an associated name:
  - `s.name = 'population'; s.index.name = 'state'`
• Addition, filtering, and NumPy operations work as expected and preserve the index-value link
• These operations align:

  ```
  In [28]: obj3
  Out[28]:
  Ohio   35000
  Oregon 16000
  Texas  71000
  Utah   5000
  dtype: int64

  In [29]: obj4
  Out[29]:
  California  NaN
  Ohio       35000
  Oregon     16000
  Texas      71000
  dtype: float64

  In [30]: obj3 + obj4
  Out[30]:
  California  NaN
  Ohio       70000
  Oregon     32000
  Texas      142000
  Utah       NaN
  dtype: float64
  ```

  [W. McKinney, Python for Data Analysis]
Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with column headers
- Has an index shared with each series
- Allows easy reference to any cell

- Index is automatically assigned just as with a series but can be passed in as well via index kwarg
- Can reassign column names by passing columns kwarg
## DataFrame Constructor Inputs

<table>
<thead>
<tr>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D ndarray</td>
<td>A matrix of data, passing optional row and column labels</td>
</tr>
<tr>
<td>dict of arrays, lists, or tuples</td>
<td>Each sequence becomes a column in the DataFrame. All sequences must be the same length.</td>
</tr>
<tr>
<td>NumPy structured/record array</td>
<td>Treated as the “dict of arrays” case</td>
</tr>
<tr>
<td>dict of Series</td>
<td>Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed.</td>
</tr>
<tr>
<td>dict of dicts</td>
<td>Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.</td>
</tr>
<tr>
<td>list of dicts or Series</td>
<td>Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels</td>
</tr>
<tr>
<td>List of lists or tuples</td>
<td>Treated as the “2D ndarray” case</td>
</tr>
<tr>
<td>Another DataFrame</td>
<td>The DataFrame’s indexes are used unless different ones are passed</td>
</tr>
<tr>
<td>NumPy MaskedArray</td>
<td>Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]