DSC 201: Data Analysis & Visualization

Series & Data Frames

Dr. David Koop
Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism
- Nesting/Composition

- Components:
  - Instance variables/methods
  - Class variables/methods
Inheritance

• class Square(Rectangle):
  def __init__(self, x, y, s):
    super().__init__(x,y,s,s)

  def set_width(self, w):
    super().set_width(w)
    super().set_height(w)

  def set_height(self, h):
    super().set_width(h)
    super().set_height(h)

• Overriding: use the same name

• No overloading, but with keyword parameters (or by checking types) can accomplish similar results if needed
Exercise: Queue Class

• Write a class to encapsulate queue behavior. It should have five methods:
  - constructor: should allow a list of initial elements (in order)
  - size: should return the number of elements
  - is_empty: returns True if the queue is empty, false otherwise
  - enqueue: adds an item to the queue
  - dequeue: removes an item from the queue and returns it
Exercise: Stack Class

• How do we modify this for a stack?
  - constructor: should allow a list of initial elements (in order)
  - size: should return the number of elements
  - is_empty: returns True if the queue is empty, False otherwise
  - push instead of enqueue: adds an item to the stack
  - pop instead of dequeue: removes an item from the stack

• Could we use inheritance?
Creating numpy (np) 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)
Operations

- (Array, Array) Operations (elementwise)
  - Addition, Subtraction, Multiplication

- (Scalar, Array) Operations:
  - Addition, Subtraction, Multiplication, Division, Exponentiation

- Slicing:
  - 1D: Just like with lists except **data is not copied!**
    - `a[2:5] = 3` works with arrays
    - `a.copy()` or `a[2:5].copy()` will copy
  - 2D+: comma separated indices as shorthand:
    - `a[1][2]` or `a[1,2]`
    - `a[1]` gives a row
    - `a[:,1]` gives a column
In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array $arr3d$:

```python
In [76]:
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```python
In [77]:
arr3d[0]
```

```
Out[77]:
array([[1, 2, 3],
       [4, 5, 6]])
```

$arr3d[0]$ is a $2 \times 3$ array:

```python
In [78]:
arr3d[0]
```

```
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Both scalar values and arrays can be assigned to $arr3d[0]$:

```python
In [79]:
old_values = arr3d[0].copy()
```

```python
In [80]:
arr3d[0] = 42
```

```python
In [81]:
arr3d
```

```
Out[81]:
array([[42, 42, 42],
       [42, 42, 42]])
```

```
[W. McKinney, Python for Data Analysis]
```
2D Array Slicing

How to obtain the blue slice from array \texttt{arr}?

[W. McKinney, Python for Data Analysis]
2D Array Slicing

How to obtain the blue slice from array \( \text{arr} \)?

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

[W. McKinney, Python for Data Analysis]
2D Array Slicing

How to obtain the blue slice from array arr?

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

How to obtain the blue slice from array \( \text{arr} \)?

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

[W. McKinney, Python for Data Analysis]
2D Array Slicing

How to obtain the blue slice from array \( \text{arr} \)?

- \( \text{arr}[:, 1:] \) (2, 2)
- \( \text{arr}[2] \) (3,)
- \( \text{arr}[2, :] \) (3,)
- \( \text{arr}[2:,: ] \) (1, 3)
- \( \text{arr}[:, :2] \) (3, 2)
- \( \text{arr}[1, :2] \) (2,)
- \( \text{arr}[1:2, :2] \) (1, 2)

[W. McKinney, Python for Data Analysis]
Assignment 3

• Create a class that processes refugee data from Assignment 1
• Requires wrangling the data
  - changing column names
  - converting strings
• Call the wrangling functions in the constructor!
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: `arr[[1, 2, 3]]`
- Transposing arrays: `arr.T`
- Reshaping arrays: `arr.reshape((3, 5))`
- Unary universal functions (ufuncs): `np.sqrt`, `np.exp`
- Binary universal functions: `np.add`, `np.maximum`
## Unary Universal Functions

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

See Tables 4-3 and 4-4 for a listing of available ufuncs.

Table 4-3. Unary ufuncs

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

Table 4-4. Binary universal functions

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

[W. McKinney, Python for Data Analysis]
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; 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>
<tr>
<td>greater, greater_equal,</td>
<td>Perform element-wise comparison, yielding boolean array (equivalent to infix</td>
</tr>
<tr>
<td>less, less_equal,</td>
<td>operators ( &gt;, \geq, \leq, &lt;, \neq, =, \neq )</td>
</tr>
<tr>
<td>equal, not_equal</td>
<td>Compute element-wise truth value of logical operation (equivalent to infix</td>
</tr>
<tr>
<td>logical_and,</td>
<td>logical_or, logical_xor</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Here, `arr.mean(1)` means “compute mean across the columns” where `arr.sum(0)` means “compute sum down the rows.” Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like `cumsum` return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
In [187]: arr.cumsum(axis=0)
Out[187]:
array([[0, 1, 2],
       [3, 5, 7],
       [6, 8, 10]])
```

```
In [188]: arr.cumprod(axis=1)
Out[188]:
array([[ 0,  0,  0],
       [ 3, 12, 60],
       [ 6, 42, 336]])
```

See Table 4-5 for a full listing. We’ll see many examples of these methods in action in later chapters.

**Table 4-5: Basic array statistical methods**

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

- Other methods:
  - any and all
  - sort
  - unique

- Linear Algebra (numpy.linalg)

- Pseudorandom Number Generation (numpy.random)
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

• Universal:
  - import pandas as pd

• Also used:
  - from pandas import Series, DataFrame
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 = pd.Series([7, 14, -2, 1])`

• Basically two arrays: `obj.values` and `obj.index`
• Can specify the index explicitly and use strings
• `obj2 = pd.Series([4, 7, -5, 3],
                   index=['d', 'b', 'a', 'c'])`

• Could think of a fixed-length, ordered dictionary
• Can create from a dictionary
• `obj3 = pd.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     35000
  Ohio           70000
  Oregon         48000
  Texas          112000
  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

```python
df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
                'pop': [1.5, 1.7, 3.6, 2.4]})
```

- 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]
DataFrame Access and Manipulation

- df.values → 2D NumPy array

- Accessing a column:
  - df["<column>"]
  - df.<column>
  - Both return Series
  - Dot syntax only works when the column is a valid identifier

- Assigning to a column:
  - df[<column>] = <scalar>  # all cells set to same value
  - df[<column>] = <array>   # values set in order
  - df[<column>] = <series>  # values set according to match
    # between df and series indexes