DSC 201: Data Analysis & Visualization

Arrays and Series

Dr. David Koop
Exception Example

- def divide(mylist, x, y):
  newlist = []
  try:
    z = x // y
    below, mid, above = \\
    mylist[:z], mylist[z], mylist[z+1:]
  except (ZeroDivisionError, IndexError):
    below, mid, above = mylist, -999, []
  else:
    newlist = below + above
  finally:
    newlist.append(-999)
Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism
- Nesting/Composition

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

- class Rectangle:
  
  ```python
  def __init__(self, x, y, w, h):
    self.x = x
    self.y = y
    self.w = w
    self.h = h
  
  def set_corner(self, x, y):
    self.x = x
    self.y = y
  
  def set_width(self, w):
    self.w = w
  
  def set_height(self, h):
    self.h = h
  
  def area(self):
    return self.w * self.h
  ```
Inheritance Example

• 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)
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.
NumPy

- Fast **vectorized** array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application).

[W. McKinney, Python for Data Analysis]
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)`
Midterm

• Tuesday, October 23, during class (2-3:15pm, entire class period)
• Material:
  - Everything since the beginning of class
  - Ch. 1-4 in book, Visualization, Exploratory Data Analysis, Python
• Format: (like Quiz 1)
  - Multiple Choice
  - Free Response
• No class on Thursday, October 25
• No office hours next week
Assignment 3

• Link
• Hurricane data, take 3
• Redo parts of A1 & A2 using pandas
• Part 5 shows how pandas can connect to altair
• May need to read ahead, but have tried to point to specific documentation for most of the concepts
• Due Thursday, Nov. 1
Types

• "But I thought Python wasn't stingy about types…"
• numpy aims for speed
• Able to do array arithmetic
• int16, int32, int64, float32, float64, bool, object
• astype method allows you to convert between different types of arrays:

```python
arr = np.array([1, 2, 3, 4, 5])
arr.dtype
float_arr = arr.astype(np.float64)
```
numpy data types (dtypes)

<table>
<thead>
<tr>
<th>Type</th>
<th>Type code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8, uint8</td>
<td>i1, u1</td>
<td>Signed and unsigned 8-bit (1 byte) integer types</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>i2, u2</td>
<td>Signed and unsigned 16-bit integer types</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>i4, u4</td>
<td>Signed and unsigned 32-bit integer types</td>
</tr>
<tr>
<td>int64, uint64</td>
<td>i8, u8</td>
<td>Signed and unsigned 64-bit integer types</td>
</tr>
<tr>
<td>float16</td>
<td>f2</td>
<td>Half-precision floating point</td>
</tr>
<tr>
<td>float32</td>
<td>f4 or f</td>
<td>Standard single-precision floating point; compatible with C float</td>
</tr>
<tr>
<td>float64</td>
<td>f8 or d</td>
<td>Standard double-precision floating point; compatible with C double and Python float object</td>
</tr>
<tr>
<td>float128</td>
<td>f16 or g</td>
<td>Extended-precision floating point</td>
</tr>
<tr>
<td>complex64,</td>
<td>c8, c16,</td>
<td>Complex numbers represented by two 32, 64, or 128 floats, respectively</td>
</tr>
<tr>
<td>complex128,</td>
<td>c32</td>
<td></td>
</tr>
<tr>
<td>complex256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>?</td>
<td>Boolean type storing True and False values</td>
</tr>
<tr>
<td>object</td>
<td>0</td>
<td>Python object type; a value can be any Python object</td>
</tr>
<tr>
<td>string_</td>
<td>S</td>
<td>Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'</td>
</tr>
<tr>
<td>unicode_</td>
<td>U</td>
<td>Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')</td>
</tr>
</tbody>
</table>

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

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 $arr3d$:

In $\begin{bmatrix} 76 \\ 77 \end{bmatrix}$:

$arr3d = \text{np.array}\left(\begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \end{bmatrix}\right)$

In $\begin{bmatrix} 77 \\ 78 \end{bmatrix}$:

$\text{out}_{\begin{bmatrix} 77 \\ 78 \end{bmatrix}}$:

$\text{array}\left(\begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \end{bmatrix}\right)$

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

In $\begin{bmatrix} 78 \\ 79 \end{bmatrix}$:

$\text{old_values} = \text{arr3d}[0]$.copy()

In $\begin{bmatrix} 79 \end{bmatrix}$:

$\text{arr3d}[0] = 42$

In $\begin{bmatrix} 80 \\ 81 \end{bmatrix}$:

$\text{out}_{\begin{bmatrix} 80 \\ 81 \end{bmatrix}}$:

$\text{array}\left(\begin{bmatrix} \begin{bmatrix} 42 & 42 & 42 \\ 42 & 42 & 42 \end{bmatrix} & \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \end{bmatrix}\right)$

In $\begin{bmatrix} 82 \end{bmatrix}$:

$\text{arr3d}[0] = \text{old_values}$

[Note: The image includes a diagram of a 2D indexing grid and references W. McKinney, Python for Data Analysis]

D. Koop, DSC 201, Fall 2018
2D Array Slicing

How to obtain the blue slice from array 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`?

### Expression | Shape
--- | ---
`arr[:2, 1:]` | `(2, 2)`
`arr[2]` | `(3,)`
`arr[2, :]` | `(3,)`
`arr[2:, :]` | `(1, 3)`

[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]])
```

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]])
```

How to obtain the blue slice from array `arr`?

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

How to obtain the blue slice from array `arr`?

Expression | Shape
---|---
`arr[:2, 1:]` | `(2, 2)`
`arr[2]` | `(3,)`
`arr[2, :]` | `(3,)`
`arr[2:, :]` | `(1, 3)`
`arr[:, :2]` | `(3, 2)`
`arr[1, :2]` | `(2,)`
`arr[1:2, :2]` | `(1, 2)`

[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: \( \text{arr.reshape((3,5))} \)
• Unary universal functions (ufuncs): \( \text{np.sqrt}, \text{np.exp} \)
• Binary universal functions: \( \text{np.add}, \text{np.maximum} \)
Unary Universal Functions

<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</td>
</tr>
<tr>
<td>sqrt</td>
<td>Compute the square root of each element (equivalent to arr ** 0.5)</td>
</tr>
<tr>
<td>square</td>
<td>Compute the square of each element (equivalent to 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 that number)</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 a 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-inf, non-NaN) or infinite, respectively</td>
</tr>
<tr>
<td>cos, cosh, sin, sinh, tan, tanh</td>
<td>Regular and hyperbolic trigonometric functions</td>
</tr>
<tr>
<td>arccos, arccosh, arcsin, arccosh, arctan, arctanh</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>logical_not</td>
<td>Compute truth value of not $x$ element-wise (equivalent to ~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>abs, fabs</td>
<td>Compute the absolute value element-wise for integer, floating-point, or complex values</td>
</tr>
<tr>
<td>sqrt</td>
<td>Compute the square root of each element (equivalent to arr ** 0.5)</td>
</tr>
<tr>
<td>square</td>
<td>Compute the square of each element (equivalent to 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 that number)</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 a 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-inf, non-NaN) or infinite, respectively</td>
</tr>
<tr>
<td>cos, cosh, sin, sinh, tan, tanh</td>
<td>Regular and hyperbolic trigonometric functions</td>
</tr>
<tr>
<td>arccos, arccosh, arcsin, arccosh, arctan, arctanh</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>logical_not</td>
<td>Compute truth value of not $x$ element-wise (equivalent to ~arr).</td>
</tr>
</tbody>
</table>

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

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; 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>

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</td>
</tr>
<tr>
<td>less, less_equal,</td>
<td>infix operators &gt;, &gt;=, &lt;, &lt;=, ==, !=)</td>
</tr>
<tr>
<td>equal, not_equal</td>
<td></td>
</tr>
<tr>
<td>logical_and,</td>
<td>Compute element-wise truth value of logical operation (equivalent to infix</td>
</tr>
<tr>
<td>logical_or, logical_xor</td>
<td>operators &amp;</td>
</tr>
</tbody>
</table>

See Tables 4-3. Unary ufuncs and Table 4-4. Binary universal functions.
**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>

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>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>

[W. McKinney, Python for Data Analysis]
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:**

```python
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
Utah        NaN
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]