

Programming Principles in Python (CSCI 503/490)

Data

Dr. David Koop

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 Arrays

- import numpy as np
- Creating:
 - `data1 = [6, 7, 8, 0, 1]`
 - `arr1 = np.array(data1)`
 - `arr1_float = np.array(data1, dtype='float64')`
 - `np.ones((4,2))` # 2d array of ones
 - `arr1_ones = np.ones_like(arr1)` # `[1, 1, 1, 1, 1]`
- Type and Shape Information:
 - `arr1.dtype` # `int64` # type of values stored in array
 - `arr1.ndim` # `1` # number of dimensions
 - `arr1.shape` # `(5,)` # shape of the array

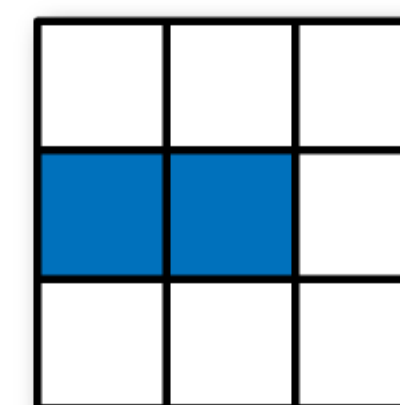
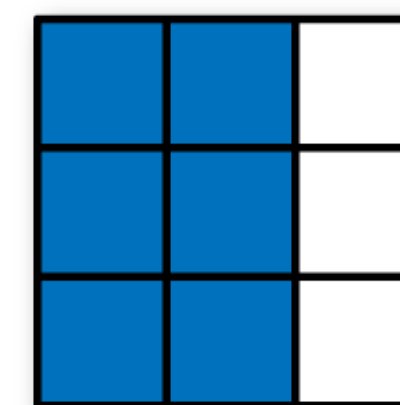
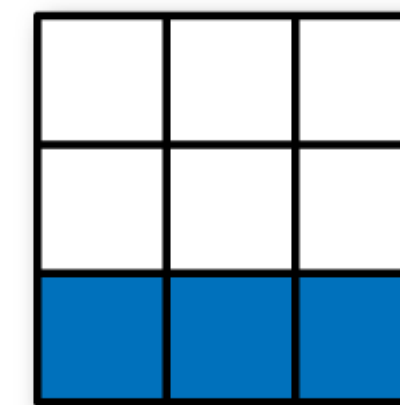
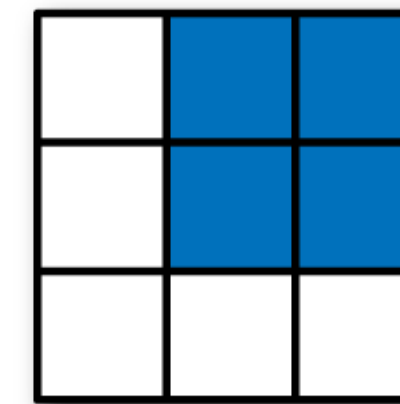
Array Operations

- `a = np.array([1, 2, 3])`
`b = np.array([6, 4, 3])`
- (Array, Array) Operations (**Element-wise**)
 - Addition, Subtraction, Multiplication
 - `a + b` # `array([7, 6, 6])`
- (Scalar, Array) Operations (**Broadcasting**):
 - Addition, Subtraction, Multiplication, Division, Exponentiation
 - `a ** 2` # `array([1, 4, 9])`
 - `b + 3` # `array([9, 7, 6])`

Indexing

- Same as with lists plus shorthand for 2D+
 - `arr1 = np.array([6, 7, 8, 0, 1])`
 - `arr1[1]`
 - `arr1[-1]`
- What about two dimensions?
 - `arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])`
 - `arr[1][1]`
 - `arr[1,1]` # shorthand

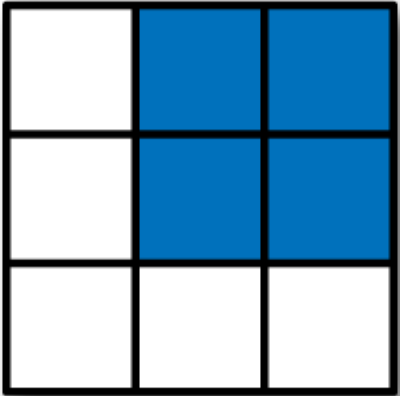
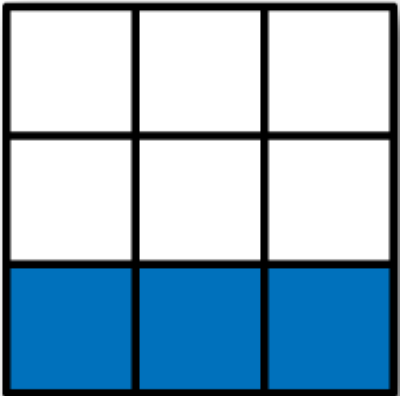
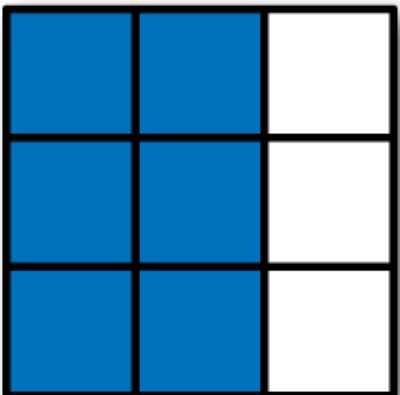
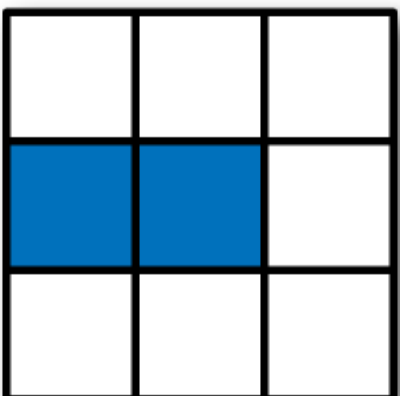
numpy Array Slicing



[W. McKinney, Python for Data Analysis]

numpy Array Slicing

- Indexing is similar to lists
 - Even in 2D
 - `arr[2][2]` same as `arr[2, 2]`
- Slicing is a bit different:
 - Slices are **views**
 - Dimensionality unchanged with pure slicing
 - `arr[1:3][:2] != arr[1:3, :2]`

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

[W. McKinney, Python for Data Analysis]

More Reshaping

- reshape:
 - `arr2.reshape(4,2)` # returns new view
- resize:
 - `arr2.resize(4,2)` # no return, modifies `arr2` in place
- flatten:
 - `arr2.flatten()` # `array([1.5, 2., 3., 4., 5., 6., 7., 8.])`
- ravel:
 - `arr2.ravel()` # `array([1.5, 2., 3., 4., 5., 6., 7., 8.])`
- flatten and ravel look the same, but ravel is a **view**

Assignment 7

- Concurrency, System Integration, and Structural Pattern Matching
- Download System Logs
- Locate Logs of Interest
- Read JSON & Binary Data
- Filter suspicious events using structural pattern matching
- Process all files using threading

Quiz Wednesday

Array Transformations

- Transpose
 - `arr2.T` # flip rows and columns
- Stacking: take iterable of arrays and stack them horizontally/vertically
 - `arrh1 = np.arange(3)`
 - `arrh2 = np.arange(3, 6)`
 - `np.vstack([arrh1, arrh2])`
 - `np.hstack([arr1.T, arr2.T])` # ???

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`

pandas

- Contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python
- Originally built on top of NumPy
- Built with the following requirements:
 - Data structures with labeled axes (aligning data)
 - Support time series data
 - Do arithmetic operations that include metadata (labels)
 - Handle missing data
 - Add merge and relational operations

polars

- Contains high-level data structures and manipulation tools designed to make data analysis "**lightning**" fast and easy in Python
 - Built using Apache Arrow
 - Written from scratch using Rust but with a Python API
 - Parallelized (uses multiple cores)
 - Intuitive API

Code Conventions

- Universal:

- `import pandas as pd`
- `import polars as pl`

- Also used:

- `from pandas import Series, DataFrame`
- `from polars import Series, DataFrame`

polars Series

- A one-dimensional data structure (with a type)
 - `s = pl.Series([1, 2, 3])`
- May also have a name
 - `s = pl.Series('name', ['a', 'b', 'c'])`
- Just like numpy arrays, a series has a dtype
 - `s = pl.Series('name', [1, 2, 3], dtype=pl.Float)`
- Indexing:
 - `s[0] # 1.0`

pandas Series

- A one-dimensional array (with a type)
 - `t = pd.Series([1, 2, 3])`
- May also have a name:
 - `t = pd.Series([1, 2, 3], name='num')`
- Just like numpy arrays, a series has a dtype
 - `t = pd.Series([1, 2, 3], name='num', dtype='float')`
- Indexing: `t[0]`
- ...but a pandas Series also has an **index** (polars does not)

pandas Series and the Index

- pandas Series is a one-dimensional array (with a type) **plus an index**
- Basically two arrays: `t.values` and `t.index`
 - `obj.index # [0, 1, 2]`
- Can specify the index explicitly (could be strings)
 - `t = pd.Series([1, 2, 3], ['a', 'b', 'c'])`
- Kind of like fixed-length, ordered dictionary + can create from a dictionary
 - `t = pd.Series({'a': 1, 'b': 2, 'c': 3})`
- Indexing:
 - `t['a']`
 - What about `t[0]`?

polars Series Operations

- Can do binary operations with two Series
- Just like numpy, between two Series, these are **elementwise**
 - `pl.Series([1,2,3]) + pl.Series([1,2,3]) # pl.Series([2,4,6])`
- Between a Series and a scalar, this is **broadcast**
 - `pl.Series([1,2,3]) + 4 # pl.Series([5,6,7])`
- Have to have the same number of elements
 - `pl.Series([1,2,3]) + pl.Series([1,2,3,4]) # Error`
- Also works with non-numeric operations:
 - `pl.Series(['a','b']) + pl.Series(['c','d'])`

pandas Series Operations

- Same as polars
 - `pd.Series([1,2,3]) + pd.Series([1,2,3]) # pd.Series([2,4,6])`
 - `pd.Series([1,2,3]) + 4 # pd.Series([5,6,7])`
- ...but with custom indexes, the operations **align**:
 - `pd.Series([1,2,3], index=list('abc')) +
pd.Series([1,2,3], index=list('cba'))
=> pd.Series([4,4,4], index=['a','b','c'])`

```
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]

pandas Series Operations

- Missing labels lead to NaN (not a number) values

```
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
```

- also have `.add`, `.subtract`, ... that allow `fill_value` argument
- `obj3.add(obj4, fill_value=0)`

DataFrame

- A collection of Series (uniquely named)
 - Similar to a table in a database
 - Similar to a sheet in a spreadsheet
- ```
df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
 'year': [2000, 2001, 2002, 2001],
 'pop': [1.5, 1.7, 3.6, 2.4]})
```
- In pandas:
  - Has an index shared with each series
  - Index is automatically assigned just as with a series but can be passed in as well via `index` kwarg



# pandas DataFrame Constructor Inputs

---

| Type                             | Notes                                                                                                                                     |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| 2D ndarray                       | A matrix of data, passing optional row and column labels                                                                                  |
| dict of arrays, lists, or tuples | Each sequence becomes a column in the DataFrame. All sequences must be the same length.                                                   |
| NumPy structured/record array    | Treated as the “dict of arrays” case                                                                                                      |
| dict of Series                   | 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. |
| dict of dicts                    | Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.                                 |
| list of dicts or Series          | Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels                       |
| List of lists or tuples          | Treated as the “2D ndarray” case                                                                                                          |
| Another DataFrame                | The DataFrame’s indexes are used unless different ones are passed                                                                         |
| NumPy MaskedArray                | Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result                                                 |

[W. McKinney, Python for Data Analysis]

# DataFrame Columns

---

- Access:
  - polars: `df['state']`
  - pandas: `dfa['state']` or `dfa.state` (doesn't always work!)
- Modification:
  - polars: `df.with_columns(pl.Series('state', ['Ohio', 'Ohio', 'Texas', 'Nevada']))`
  - pandas: `df.assign(state=['Ohio', 'Ohio', 'Texas', 'Nevada'])`
  - Both create **new** data frames
  - pandas: `df['state'] = ['Ohio', 'Ohio', 'Texas', 'Nevada']`
  - This **mutates** the dataframe but causes problems so avoid it!



# DataFrame Multiple Columns

---

- polars:
  - `df.select('state', 'year')`
- pandas:
  - `df[['state', 'year']]`
  - Not a new operator! It is a subscript where the argument is a list

# DataFrame Indexing and Slicing

---

- polars:
  - `df[0]`, `df[0:1]` # equivalent, data frame with single row
- pandas:
  - `dfa[0]` # error
  - `dfa.loc[0]` # a Series!
  - `dfa[0:2]` # a data frame with two rows
- pandas with an index (`dfi = dfa.set_index('state')`)
  - `dfi['Texas']`, `dfi['Ohio']` # a Series, a DataFrame!
  - `dfi.loc['Ohio':'Texas']` # inclusive slice!
  - `dfi.iloc[0:2]` # not inclusive!

# pandas DataFrame Indexing and Slicing

---

- Same as with NumPy arrays but can use index labels
- Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  - `s = Series(np.arange(4))`  
`s[0:2]` # gives two values like numpy
  - `s = Series(np.arange(4), index=['a', 'b', 'c', 'd'])`  
`s['a':'c']` # gives three values, not two!
- Obtaining data subsets
  - `loc`: get rows/cols by label
  - `iloc`: get rows/cols by position (integer index)