DSC 201: Data Analysis & Visualization

Classes & Arrays

Dr. David Koop
Sets

- Sets are like dictionaries but without any values:
  - \( s = \{ 'MA', 'RI', 'CT', 'NH' \}; \ t = \{ 'MA', 'NY', 'NH' \} \)
- \{\} is an empty dictionary, \( \text{set}() \) is an empty set
- Adding values: \( s.\text{add}(\text{\'ME\'}) \)
- Removing values: \( s.\text{discard}(\text{\'CT\'}) \)
- Exists: "CT" in \( s \)
- Union: \( s \ | \ t \Rightarrow \{ 'MA', 'RI', 'CT', 'NH', 'NY' \} \)
- Intersection: \( s \ & \ t \Rightarrow \{ 'MA', 'NH' \} \)
- Exclusive-or (xor): \( s ^ t \Rightarrow \{ 'RI', 'CT', 'NY' \} \)
- Difference: \( s - t \Rightarrow \{ 'RI', 'CT' \} \)
None

- Like null in other languages
- Used as a placeholder when no value exists
- The value returned from a function that doesn't return a value:
  ```python
def f(name):
    print("Hello,", name)
    v = f("Patricia") # v will have the value None
  ```
- Also used when you need to create a new list or dictionary:
  ```python
def add_letters(s, d=None):
    if d is None:
        d = {}
    d.update(count_letters(s))
  ```
- Looks like `d={} would make more sense, but that causes issues
- `None serves as a sentinel value in add_letters`
is and ==

• == does a normal equality comparison
• is checks to see if the object is the exact same object
• Common style to write statements like if d is None: ...

• Weird behavior:
  - a = 4 - 3
    a is 1 # True
  - a = 10 ** 3
    a is 1000 # False
  - a = 10 ** 3
    a == 1000 # True

• Python caches common integer objects
• Generally, avoid is unless writing is None
Python Modules

• Python module: a file containing definitions and statements

• Import statement: like Java, get a module that isn't a Python builtin

    import collections
    d = collections.defaultdict(list)
    d[3].append(1)

• From...import...: don't need to refer to the module

    from collections import defaultdict
    d = defaultdict(list)
    d[3].append(1)
Comprehensions

- Shorthand for transformative or filtering for loops

```python
squares = []
for i in range(10):
    if i % 3 != 1:
        squares.append(i ** 2)
```

- `squares = [i**2 for i in range(10) if i % 3 != 1]`

- Equivalent code, just moved the loop inside of list definition

- Advantages: concise, readable

- Also works for dictionaries

```python
names = {"Al": ["Smith", "Brown"], "Beth": ["Jones"]}
first_counts = {k: len(v) for k, v in names.items()}
```
Assignment 3

• Not posted yet
Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism
- Nesting/Composition

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

```python
• class Rectangle:
  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
```
Advanced Classes

• Can have class variables (defined in the class body)
• Can have class methods (use `@classmethod`)
• Can have static methods (use `@staticmethod`)
• Class methods are passed the class as an object, static methods take no instance or class arguments
• Can have properties (use `@property`, `<name> .setter`)
• `@` directives are called **decorators** and precede the class/method they decorate
Advanced Classes

• A class variables and instance variable can have the same name...

• class A:
  
  b = 6
  
  def set_b(self):
    self.b = 3

  @classmethod
  def set_class_b(cls):
    cls.b = 7

• a = A()
  a.b # 6
  a.set_b()
  a.b # 3
  a.set_class_b()
  a.b # 3
  A.b # 7
Properties

- class Rectangle:
  
  ```python
  def __init__(self, x, y, w, h):
      self.x = x
      self.y = y
      self.w = w
      self.h = h
  
  @property
  def width(self):
      return self.w
  
  @width.setter
  def width(self, w):
      if w > 0: self.w = w
  
  r = Rectangle(0,0,12,3)
  print(r.width)
  r.width = 4
  ```
Inheritance

• Parentheses after the class name indicate superclass

• class Rectangle:
  def __init__(self, x, y, w, h):
    self.x = x; self.y = y
    self.w = w; self.h = h

class Square(Rectangle):
  def __init__(self, x, y, s):
    self.x = x; self.y = y
    self.w = s; self.h = s

• super() can be used to call the superclass method:

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

• Python allows multiple inheritance (multiple classes separated by commas in the parentheses)
Overriding and Overloading Methods

- 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?
Arrays

What is the difference between an array and a list (or a tuple)?
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.
Why 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]
import numpy as np
Notebook

- ch04.ipynb
- Click the raw button and save that file to disk
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)`
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
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 $\text{arr3d}$:

$\text{In } [\text{76}]: \text{arr3d} = \text{np.array}([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])$

$\text{Out } [\text{77}]: \text{array}([[\text{1, 2, 3}], [\text{4, 5, 6}], [\text{7, 8, 9}], [\text{10, 11, 12}]])$

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

$\text{In } [\text{78}]: \text{arr3d}[0]$

$\text{Out } [\text{79}]: \text{array}([[\text{1, 2, 3}], [\text{4, 5, 6}]])$

Both scalar values and arrays can be assigned to $\text{arr3d}[0]$:

$\text{In } [\text{80}]: \text{old_values} = \text{arr3d}[0].\text{copy}()$

$\text{In } [\text{81}]: \text{arr3d}[0] = 42$

$\text{Out } [\text{82}]: \text{array}([[\text{42, 42, 42}], [\text{42, 42, 42}], [\text{7, 8, 9}], [\text{10, 11, 12}]])$

$\text{arr3d}[0] = \text{old_values}$
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 `arr`?

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

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

[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: `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>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 <code>arr ** 0.5</code>)</td>
</tr>
<tr>
<td>square</td>
<td>Compute the square of each element (equivalent to <code>arr ** 2</code>)</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, arccsinh, arctan, arctanh</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>logical_not</td>
<td>Compute truth value of $\text{not } x$ element-wise (equivalent to $\sim \text{arr}$)</td>
</tr>
</tbody>
</table>
Now, evaluating the function is a matter of writing the same expression you would...

...powerful method for vectorizing computations. As a simple example, suppose we wished to evaluate the function...produces two 2D matrices corresponding to all pairs of...of magnitude faster than their pure Python equivalents, with the biggest impact in...

---

**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>Compute element-wise truth value of logical operation (equivalent to infix</td>
</tr>
<tr>
<td>logical_and,</td>
<td>operators &amp;</td>
</tr>
<tr>
<td>logical_or, logical_xor</td>
<td></td>
</tr>
</tbody>
</table>

---

[W. McKinney, Python for Data Analysis]
Here, \( \text{arr.mean(1)} \) means "compute mean across the columns" where \( \text{arr.sum(0)} \) means "compute sum down the rows."

Other methods like \text{cumsum} and \text{cumprod} do not aggregate, instead producing an array of the intermediate results:

In [184]:
\text{arr} = \text{np.array([0, 1, 2, 3, 4, 5, 6, 7])}

In [185]:
\text{arr.cumsum}()

Out [185]:
\text{array([ 0,  1,  3,  6, 10, 15, 21, 28])}

In multidimensional arrays, accumulation functions like \text{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]:
\text{arr} = \text{np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])}

In [187]:
\text{arr.cumsum(axis=0)}

Out [187]:
\text{array([[ 0,  1,  2],
            [ 3,  5,  7],
            [ 9, 11, 15]])}

\text{arr.cumprod(axis=1)}

Out [188]:
\text{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>
<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`)