Advanced Data Management (CSCI 490/680)

Structured Data

Dr. David Koop
Objects

- `d = dict()` # construct an empty dictionary object
- `l = list()` # construct an empty list object
- `s = set()` # construct an empty set object
- `s = set([1,2,3,4])` # construct a set with 4 numbers

- Calling methods:
  - `l.append('abc')`
  - `d.update({"a": 'b'})`
  - `s.add(3)`

- The method is tied to the object preceding the dot (e.g. `append` modifies `l` to add `"abc"`)
Python Modules

• Python module: a file containing definitions and statements

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

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

• `import <name> as <shorter-name>`

  ```python
  import collections as c
  ```

• `from <module> import <name>` – don't need to refer to the module

  ```python
  from collections import defaultdict
d = defaultdict(list)
d[3].append(1)
  ```
Other Collections Features

• `collections.defaultdict`: specify a default value for any item in the dictionary (instead of `KeyError`)

• `collections.OrderedDict`: keep entries ordered according to when the key was inserted
  - `dict` objects are ordered in Python 3.7 but `OrderedDict` has some other features (equality comparison, reversed)

• `collections.Counter`: counts hashable objects, has a `most_common` method
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 as a **sentinel** value when you need to create a new object:
  
  ```python
def add_letters(s, d=None):
    if d is None:
      d = {}
    d.update(count_letters(s))
    return d
```

- Looks like `d={}` would make more sense, but that causes issues
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]` 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 Comprehensions

• Shorthand for transformative or filtering for loops

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

• squares = [i**2 for i in range(10)]

• Filtering:

• 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]

• if clause follows the for clause
Dictionary Comprehensions

• Similar idea, but allow dictionary construction
• Could use lists:
  
  - names = dict([(k, v) for k, v in ... if ...])

• Native comprehension:
  
  - names = {"Al": ["Smith", "Brown"], "Beth": ["Jones"]}
  
  first_counts = {k: len(v) for k, v in names.items()}

• Could do this with a for loop as well
Assignment 1

• Using Python for data analysis
• Analyze hurricane data (through 2018)
• Provided a1.ipynb file (right-click and download)
• Use basic python (+ collections module) for now to demonstrate language knowledge
• Use Anaconda or hosted Python environment
• Due next Wednesday
• Turn .ipynb file in via Blackboard
Exceptions

- errors but potentially something that can be addressed
- try-except-else-finally:
  - except clause runs if exactly the error(s) you wish to address happen
  - else clause will run if no exceptions are encountered
  - finally always runs (even if the program is about to crash)
- Can have multiple except clauses
- can also raise exceptions using the raise keyword
- (and define your own)
Classes

- class ClassName:
  ...
- Everything in the class should be indented until the declaration ends
- self: this in Java or C++ is self in Python
- Every instance method has self as its first parameter
- Instance variables are defined in methods (usually constructor)
- __init__: the constructor, should initialize instance variables
- def __init__(self):
  self.a = 12
  self.b = 'abc'
- def __init__(self, a, b):
  self.a = a
  self.b = b
Class Example

- 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
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
Textbook's Notebooks

- ch04.ipynb
- Click the raw button and save that file to disk
- …or download/clone the entire repository
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)
Creating Arrays

- 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

• Indexing
  - Same as with lists plus shorthand for 2D+
  - `arr = np.array([[1,2],[3,4]])`
  - `arr[1,1]`
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 2×2×3 array `arr3d`:

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

```
arr3d[0] is a 2×3 array:
```
```
old_values = arr3d[0].copy()
```
```
arr3d[0] = 42
```
```
arr3d Out:
array([[42, 42, 42],
       [42, 42, 42],
       [ 7,  8,  9],
       [10, 11, 12]])
```

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

```
```

---

[W. McKinney, Python for Data Analysis]
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 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><code>arr[:2, 1:]</code></td>
<td>(2, 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>
</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><code>arr[:, 1:]</code></td>
<td>(2, 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>

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

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

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

[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 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, isninf</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, arcsinh, 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>
## 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;, &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>

In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in array computations. Later, in Table 4-3, a listing of available ufuncs.

Table 4-3. Unary ufuncs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal, not_equal</td>
<td>Compute element-wise truth value of logical operation (equivalent to infix operators &amp;</td>
</tr>
</tbody>
</table>

Add corresponding elements in arrays

Subtract elements in second array from first array

Multiply array elements

Divide or floor divide (truncating the remainder)

Raise elements in first array to powers indicated in second array

Element-wise maximum; fmax ignores NaN

Element-wise minimum; fmin ignores NaN

Element-wise modulus (remainder of division)

Copy sign of values in second argument to values in first argument

Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)

Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

[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],
       [ 9, 12, 15]])
```

In [188]:
```
arr.cumprod(axis=1)
```

Out [188]:
```
array([[ 0,  0,  0],
       [ 3, 12, 40],
       [ 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)
Chicago Food Inspections Exploration

• Based on David Beazley's PyData Chicago talk
• YouTube video: https://www.youtube.com/watch?v=j6VSAsKAj98
• Our in-class exploration:
  - Don't focus on the syntax
  - Focus on:
    • What is information is available
    • **Questions** are interesting about this dataset
    • How to decide on good follow-up questions
    • What the computations mean
Chicago Food Inspections Exploration