DSC 201: Data Analysis & Visualization

Arrays

Dr. David Koop
• 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
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:

  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?
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]` and `range(100)[-1]` 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)]
  
- Equivalent code, just moved the loop inside of list definition
  
- Advantages: concise, readable
  
  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
• `dict([(k, v) for k,v in ... if ...])`
• `{k: v for k, v in ... if ...}`
• Could do this with a for loop as well
Exceptions

- errors but potentially something that can be addressed
- try-except: except allows specifications of exactly the error(s) you wish to address

- finally: always runs (even if the program is about to crash)
- can also raise exceptions using the raise keyword
- …and define your own
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)
  ```
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 Array Slicing

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

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:

```python
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:

```python
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):

```python
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

[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`