Advanced Data Management (CSCI 490/680)

Structured Data

Dr. David Koop

Python Containers

- Container: store more than one value
- Mutable versus immutable: Can we update the container?
  - Yes → mutable
  - No → immutable
  - Lists are mutable, tuples are immutable
- Lists and tuples may contain values of different types:
- List: `[1,"abc",12.34]`
- Tuple: `(1, "abc", 12.34)`
- You can also put functions in containers!
- `len` function: number of items: `len(l)`

Indexing and Slicing

- Just like with strings
- Indexing:
  - Where do we start counting?
  - Use brackets `[]` to retrieve one value
  - Can use negative values (count from the end)
- Slicing:
  - Use brackets plus a colon to retrieve multiple values:

    `[<start>:<end>]`
  - Returns a **new** list (`b = a[:]`)
  - Don't need to specify the beginning or end

Dictionaries

- One of the most useful features of Python
- Also known as associative arrays
- Exist in other languages but a core feature in Python
- Associate a key with a value
- When I want to find a value, I give the dictionary a key, and it returns the value
- Example: InspectionID (key) → InspectionRecord (value)
- Keys must be immutable (technically, hashable):
  - Normal types like numbers, strings are fine
  - Tuples work, but lists do not (TypeError: unhashable type: 'list')
- There is only one value per key!

Sets

- Sets are like dictionaries but without any values:
- `s = {'MA', 'RI', 'CT', 'NH'}; t = {'MA', 'NY', 'NH'}`
- `{}` is an empty dictionary, `set()` is an empty set
- Adding values: `s.add('ME')`
- Removing values: `s.discard('CT')`
- Exists: `"CT" in s`
- Union: `s | t => {'MA', 'RI', 'CT', 'NH', 'NY'}`
- Intersection: `s & t => {'MA', 'NH'}`
- Exclusive-or (xor): `s ^ t => {'RI', 'CT', 'NY'}`
- Difference: `s - t => {'RI', 'CT'}`

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

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

- `import <name> as <shorter-name>`

```
import collections as c
```

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

```
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

Assignment 1

- Due Monday, Feb. 1 at 11:59pm
- Using Python for data analysis on Info Wanted ads
- Provided a1.ipynb file (right-click and download)
- Use basic python for now to demonstrate language knowledge
  - No pandas (for now)
- Use Anaconda or hosted Python environment
- Turn .ipynb file in via Blackboard
- Notes:
  - Bug in URL (https instead of http)
  - Be careful with extra spaces

Iterators

- Remember `range, values, keys, items`?
- They return **iterators**: objects that traverse containers
- 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
      ```

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

- https://github.com/wesm/pydata-book/
- 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:

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

numpy data types (dtypes)

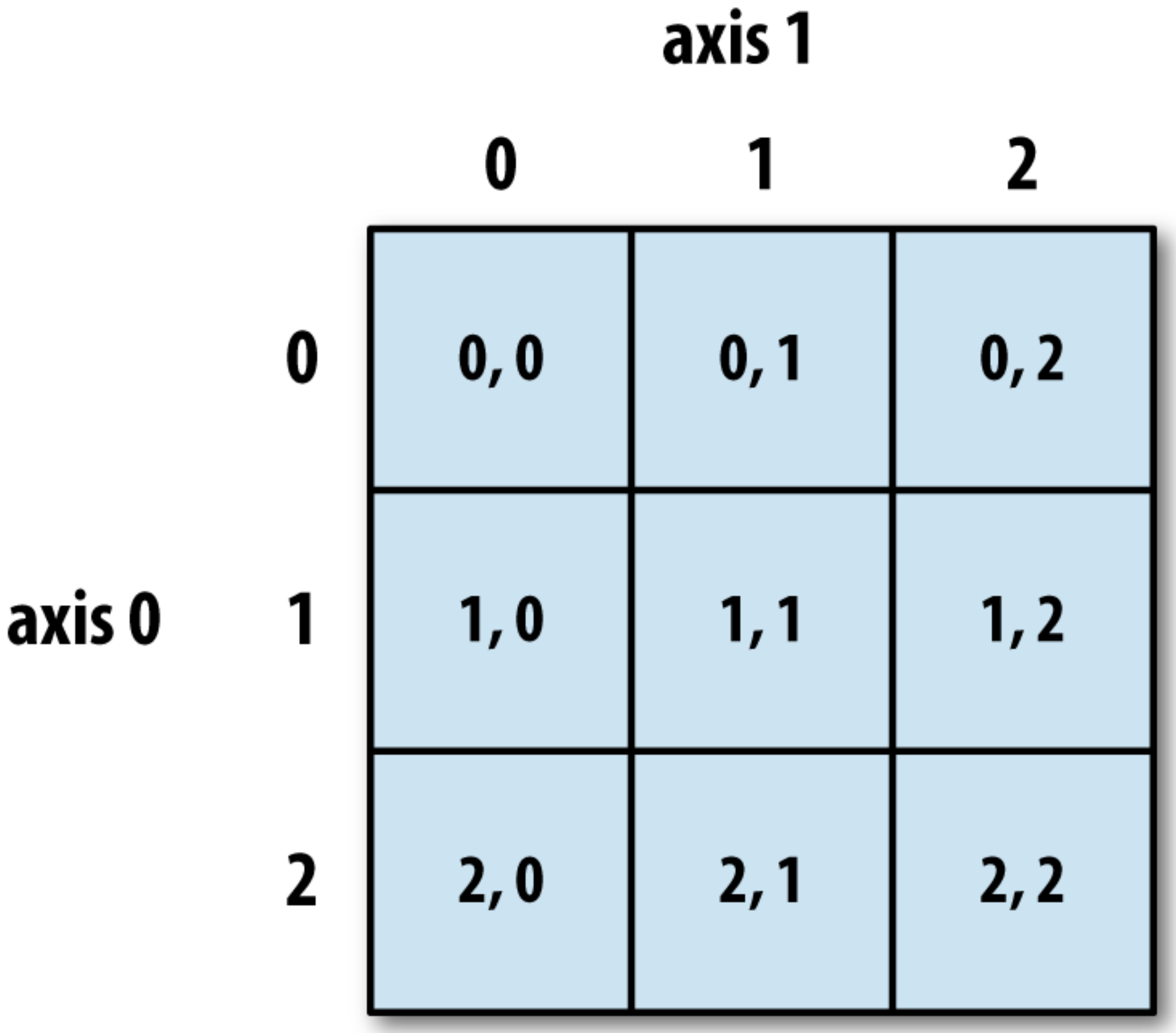
Type	Type code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 64-bit integer types
`float16`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

[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]
    ```

2D Indexing



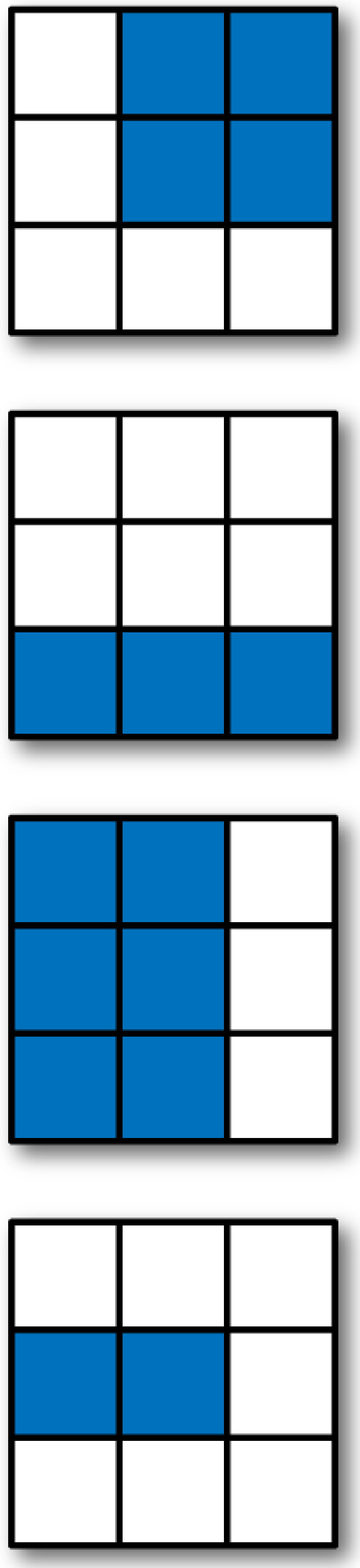
[W. McKinney, Python for Data Analysis]

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

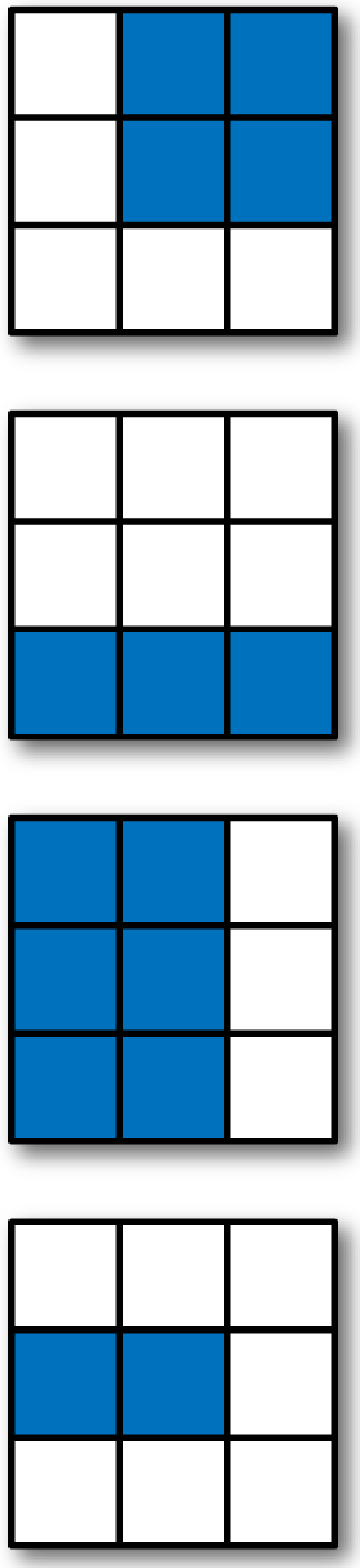
How to obtain the blue slice from array `arr`?



[W. McKinney, Python for Data Analysis]

2D Array Slicing

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

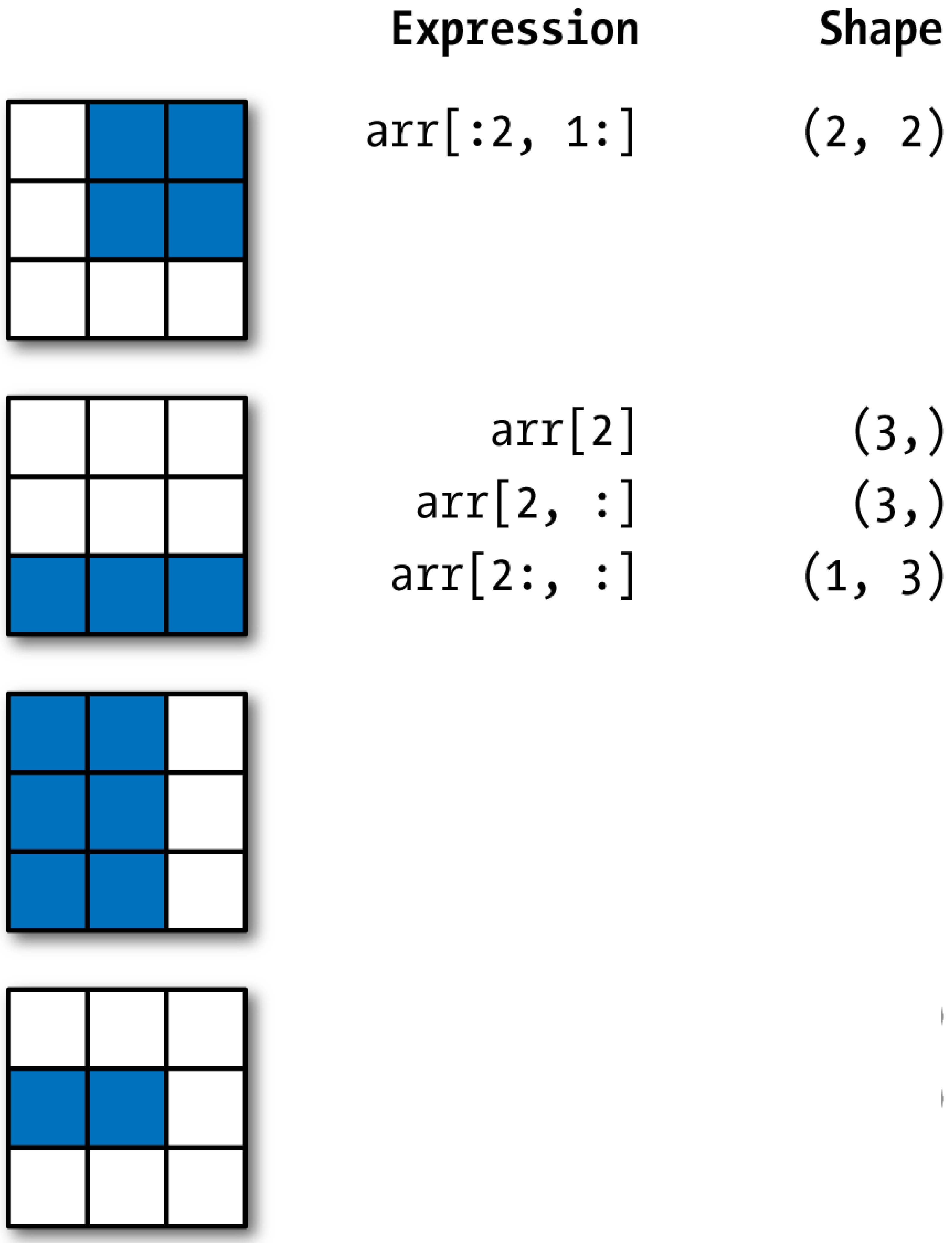


Expression	Shape
`arr[:2, 1:]`	(2, 2)

[W. McKinney, Python for Data Analysis]

2D Array Slicing

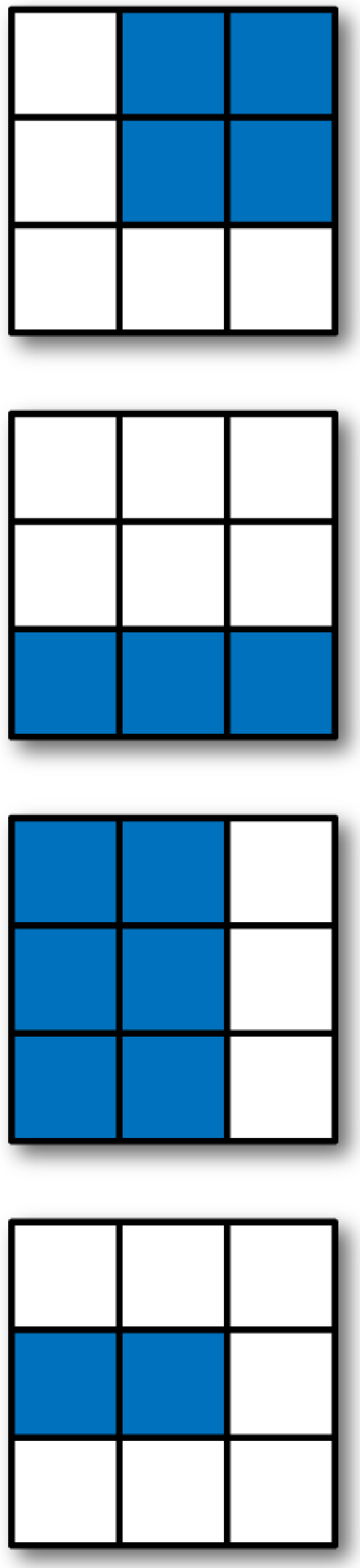
How to obtain the blue slice from array `arr`?



[W. McKinney, Python for Data Analysis]

2D Array Slicing

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



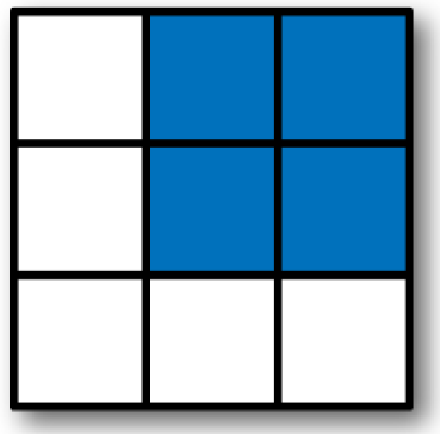
Expression	Shape
`arr[:2, 1:]`	(2, 2)
`arr[2]`	(3,)
`arr[2, :]`	(3,)
`arr[2:, :]`	(1, 3)
`arr[:, :2]`	(3, 2)

[W. McKinney, Python for Data Analysis]

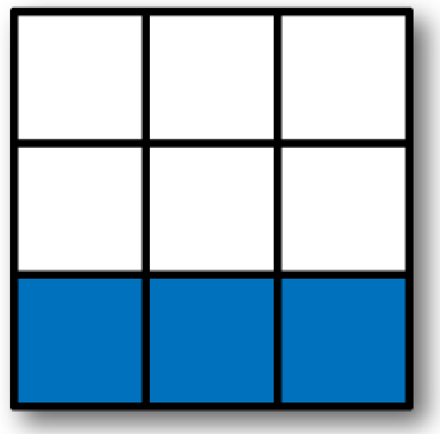
2D Array Slicing

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

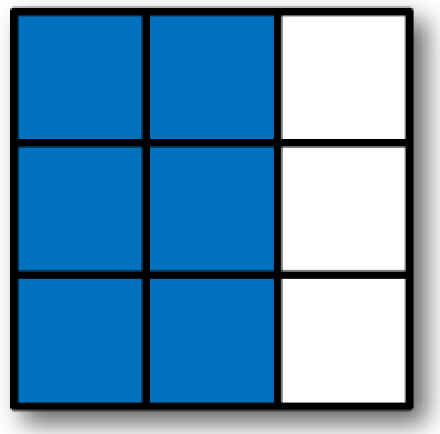
Expression	Shape



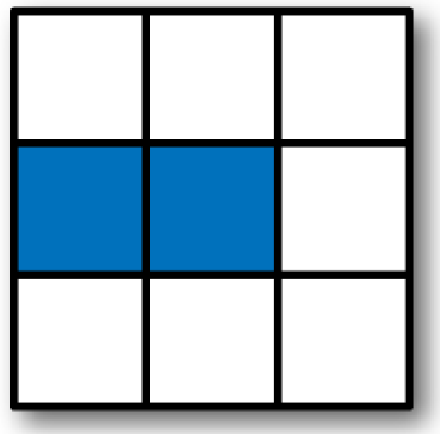
Expression	Shape
`arr[:2, 1:]`	(2, 2)



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



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



Expression	Shape
`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`