Advanced Data Management (CSCI 490/680)

Structured Data

Dr. David Koop

Components of SQL

- **Data Definition Language (DDL)**: the specification of information about relations, including schema, types, integrity constraints, indices, storage
- **Data Manipulation Language (DML)**: provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**: the DDL includes commands for specifying integrity constraints.
- **View definition**: The DDL includes commands for defining views.
- Also: **Transaction control**, **embedded and dynamic SQL**, **authorization**

[A. Silberschatz et al.]

Create Table

- An SQL relation is defined using the create table command:

  `create table r` $(A_1\ D_1,\ A_2\ D_2,\ \ldots,\ A_n\ D_n,\ (C_1),\ \ldots,\ (C_k))$

  - `r` is the **name** of the relation
  - each A_i is an **attribute name** in the schema of relation `r`
  - D_i is the **data type** of values in the domain of attribute A_i
- Example:

```
create table instructor(
        ID              char(5),
        name            varchar(20),
        dept_name       varchar(20),
        salary          numeric(8,2));
```

C_i are integrity constraints: keys, foreign keys

[A. Silberschatz et al.]

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an **attribute**
  - r_i represents a **relation**
  - P is a **predicate**.

- The result of an SQL query is a **relation**

[A. Silberschatz et al.]

Select

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: Find the names of all instructors
  - ```
    select name
    from instructor;
    ```
- Note: SQL names are **case insensitive**
  - `Name` and `NAME` and `name` are equivalent
  - Some people use upper case for language keywords (e.g. `SELECT`)

[A. Silberschatz et al.]
    ```

Where

- The operands can be expressions with operators <, <=, >, >=, =, and **<>**
- SQL allows the use of the logical connectives `and`, `or`, and `not`
- Comparisons can be applied to results of arithmetic expressions
- Example: Find all instructors in Comp. Sci. with salary > 70000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000
```

name
Katz
Brandt

[A. Silberschatz et al.]

From

- The **from** clause lists the relations involved in the query
  - Corresponds to the **Cartesian Product** operation in relational algebra
- Find the Cartesian product `instructor` X `teaches`
  - ```
    select *
    from instructor, teaches;
    ```
  - All possible `instructor` – `teaches` pair, with all attributes from both
  - Shared attributes (e.g., `ID`) are renamed (e.g., `instructor.ID`)
- Not very useful directly but useful combined with where clauses.

[A. Silberschatz et al.]
    ```

Group By

- Find the average salary of instructors in each department
  - ```
    select dept_name, avg(salary) as avg_salary
    from instructor
    group by dept_name;
    ```

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

[A. Silberschatz et al.]
    ```

Deletion

- Delete all instructors: **delete from** `instructor;`
- Delete all instructors from the Finance department
  - ```
    delete from instructor
    where dept_name= 'Finance';
    ```
- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building
  - ```
    delete from instructor
    where dept_name in (select dept_name
                        from department
                        where building = 'Watson');
    ```

[A. Silberschatz et al.]

Insertion

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.
  - ```
    insert into instructor
       select ID, name, dept_name, 18000
       from student
       where dept_name = 'Music' and total_cred > 144;
    ```
- The select-from-where statement is evaluated fully before any of its results are inserted into the relation.
- If not queries like

```
insert into table1 select * from table1
```

would cause problems

[A. Silberschatz et al.]
```

Updates

- Give a 5% salary raise to all instructors
  - ```
    update instructor
    set salary = salary * 1.05
    ```
- Give a 5% salary raise to those instructors who earn less than 70000
  - ```
    update instructor
    set salary = salary * 1.05
    where salary < 70000;
    ```
- Give a 5% salary raise to instructors whose salary is less than average
  - ```
    update instructor
    set salary = salary * 1.05
    where salary < (select avg(salary) from instructor);
    ```

[A. Silberschatz et al.]
    ```

Joins

- Join operations take two relations and return another relation.
- From relational algebra, this is a Cartesian product + selection
- Want tuples in the two relations to match (under some condition)
- The join operations typically used as subquery expressions in the from clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

[A. Silberschatz et al.]

Join Examples

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

prereq

Left Join

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Join

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

[A. Silberschatz et al.]

Join Examples

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

`course`

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

`prereq`

(Full) Outer Join

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Inner Join

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

[A. Silberschatz et al.]

Assignment 1

- Due Today at 11:59pm
- Using Python for data analysis on the Met's artwork
- 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:
  - You will need to do some parsing of the data (converting to ints, splitting strings)

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
```

PyData 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, 8, 0, 1]`
  `arr1 = np.array(data1)`
- `data2 = [[1.5,2,3,4],[5,6,7,8]]`
  `arr2 = np.array(data2)`
- `data3 = np.array([6, "abc", 3.57]) # !!! check !!!`
- Can check the type of an array in `dtype` property
- Types:
  - `arr1.dtype # dtype('int64')`
  - `arr3.dtype # dtype('<U21'), unicode plus # chars`

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
- Can specify type explicitly
  - `arr1_float = np.array(data1, dtype='float64')`
- `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]

Speed Benefits

- Compare random number generation in pure Python versus numpy
- Python:
  - ```
    import random
    %timeit rolls_list = [random.randrange(1,7)
                          for i in range(0, 60_000)]
    ```
- With NumPy:
  - ```
    %timeit rolls_array = np.random.randint(1, 7, 60_000)
    ```
- Significant speedup (80x+)

Array Shape

- Our normal way of checking the size of a collection is… `len`
- How does this work for arrays?
- ```
  arr1 = np.array([1,2,3,6,9])
  len(arr1) # 5
  ```
- ```
  arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])
  len(arr2) # 2
  ```
- All dimension lengths → shape: `arr2.shape # (2,4)`
- Number of dimensions: `arr2.ndim # 2`
- Can also reshape an array:
  - `arr2.reshape(4,2)`
  - `arr2.reshape(-1,2) # what happens here?`

Array Programming

- Lists:
  - ```
    c = []
    for i in range(len(a)):
        c.append(a[i] + b[i])
    ```
- How to improve this?
    ```

Array Programming

- Lists:
  - ```
    c = []
    for i in range(len(a)):
        c.append(a[i] + b[i])
    ```
  - ```
    c = [aa + bb for aa, bb in zip(a,b)]
    ```
- NumPy arrays:
  - ```
    c = a + b
    ```
- More functional-style than imperative
- **Internal iteration** instead of external
    ```

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])`

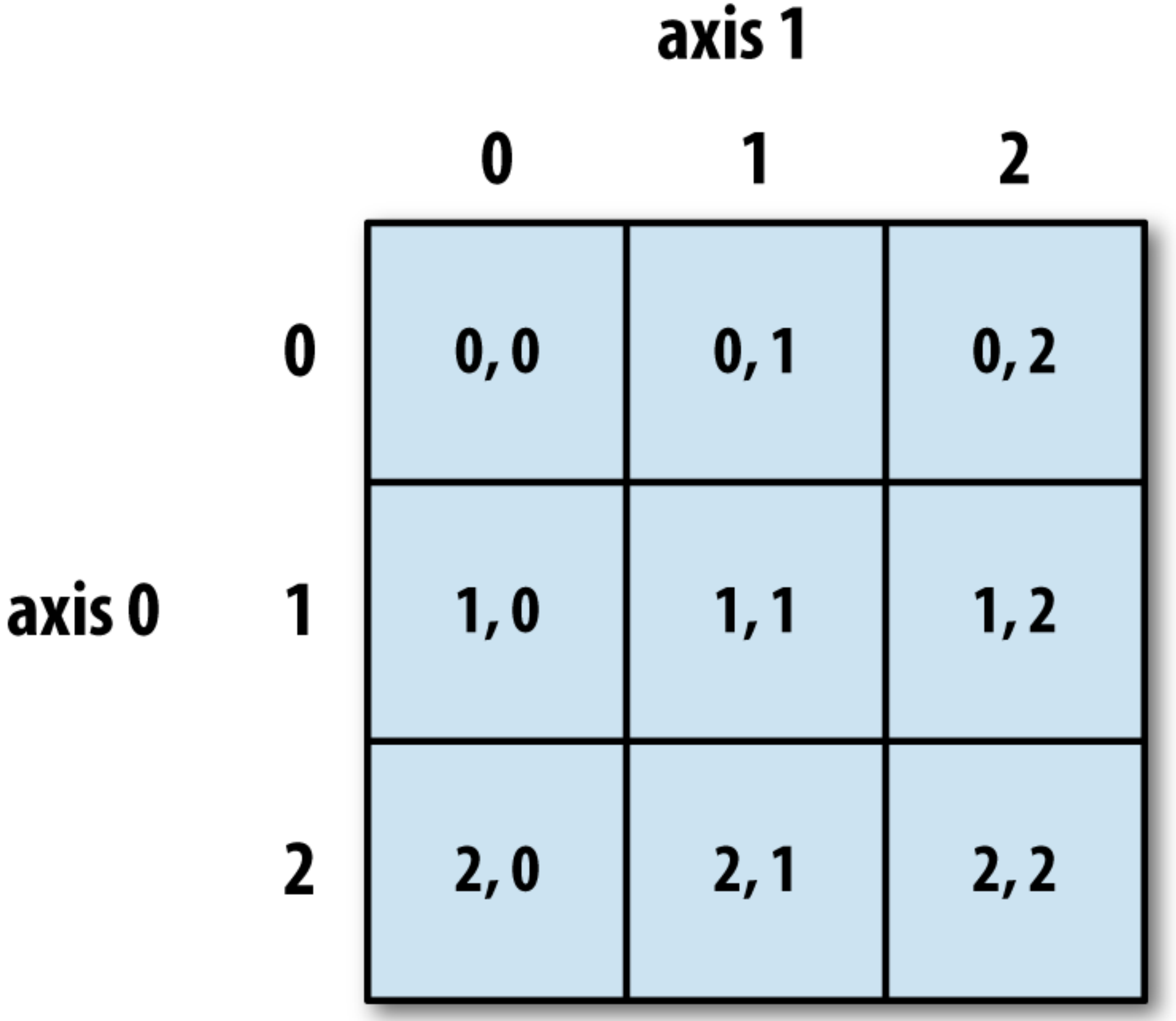
More on Array Creation

- Zeros: `np.zeros(10)`
- Ones: `np.ones((4,5)) # shape`
- Empty: `np.empty((2,2))`
- _like versions: pass an existing array and matches shape with specified contents
- Range: `np.arange(15) # constructs an array, not iterator!`

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`

2D Indexing



[W. McKinney, Python for Data Analysis]

Slicing

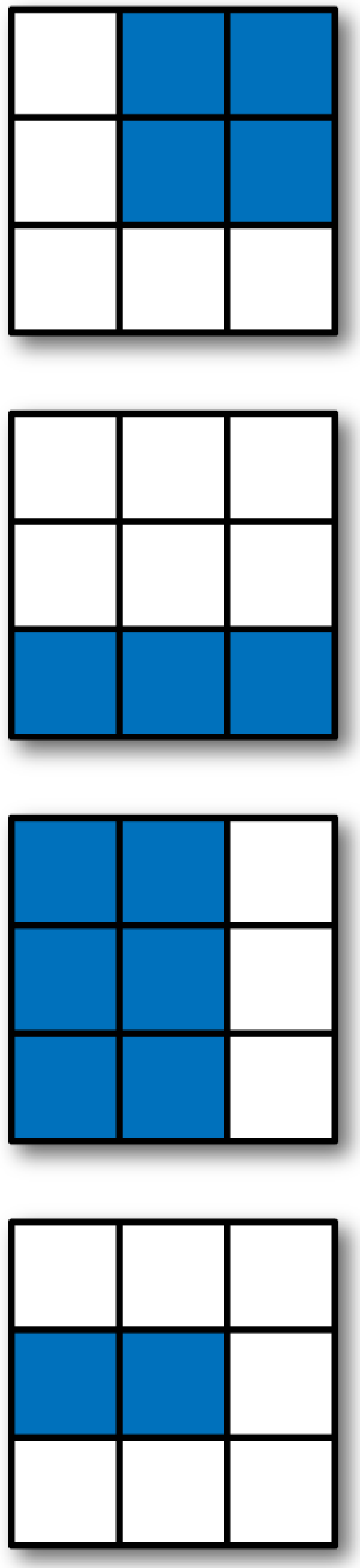
- 1D: Similar to lists
  - `arr1 = np.array([6, 7, 8, 0, 1])`
  - `arr1[2:5] # np.array([8,0,1]), sort of`
- Can **mutate** original array:
  - `arr1[2:5] = 3 # supports assignment`
  - `arr1 # the original array changed`
- Slicing returns **views** (copy the array if original array shouldn't change)
  - `arr1[2:5] # a view`
  - `arr1[2:5].copy() # a new array`

Slicing

- 2D+: comma separated indices as shorthand:
  - `arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])`
  - `a[1:3,1:3]`
  - `a[1:3,:] # works like in single-dimensional lists`
- Can combine index and slice in different dimensions
  - `a[1,:] # gives a row`
  - `a[:,1] # gives a column`

2D Array Slicing

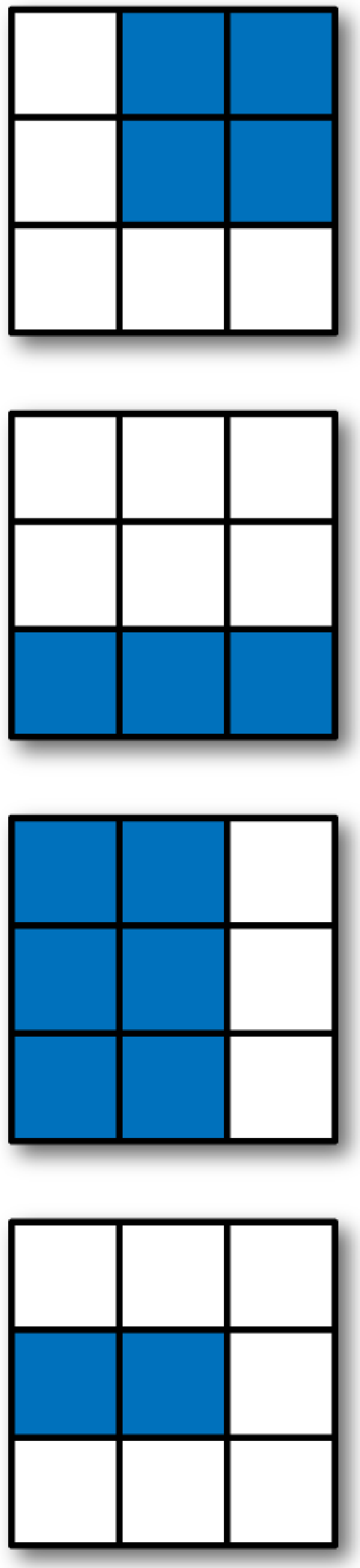
How to obtain the blue slice from array `arr`?



[W. McKinney, Python for Data Analysis]

2D Array Slicing

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

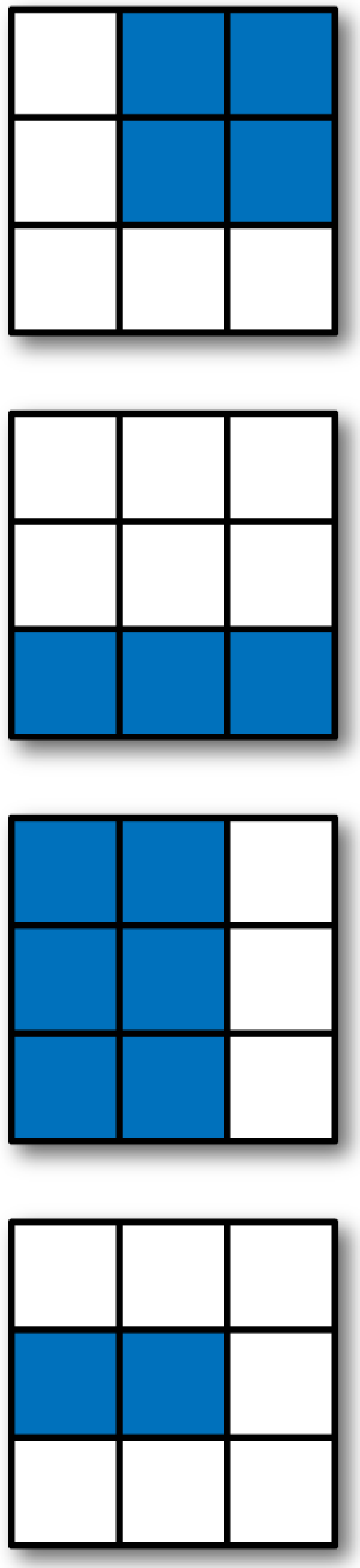


Expression	Shape
`arr[:2, 1:]`	(2, 2)

[W. McKinney, Python for Data Analysis]

2D Array Slicing

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

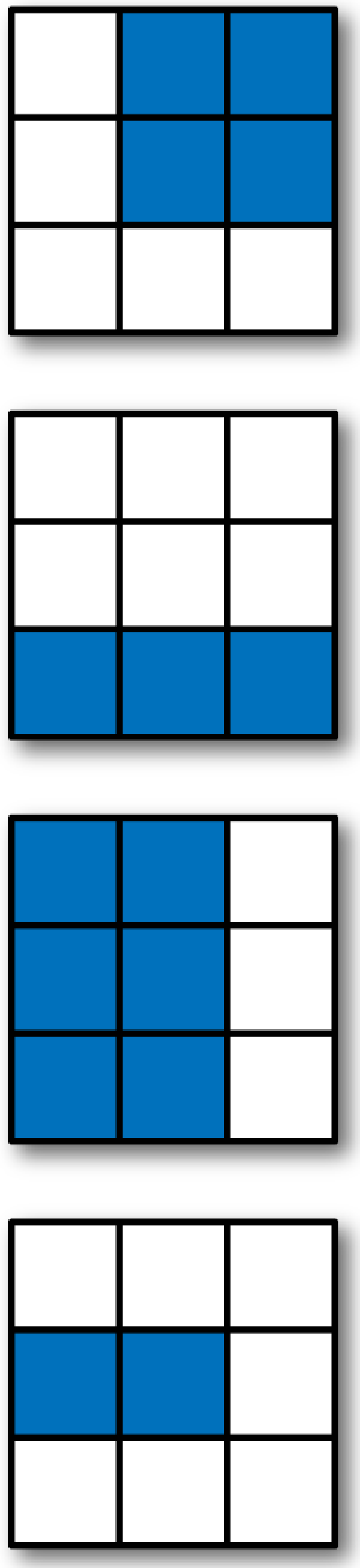


Expression	Shape
`arr[:2, 1:]`	(2, 2)
`arr[2]`	(3,)
`arr[2, :]`	(3,)
`arr[2:, :]`	(1, 3)

[W. McKinney, Python for Data Analysis]

2D Array Slicing

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

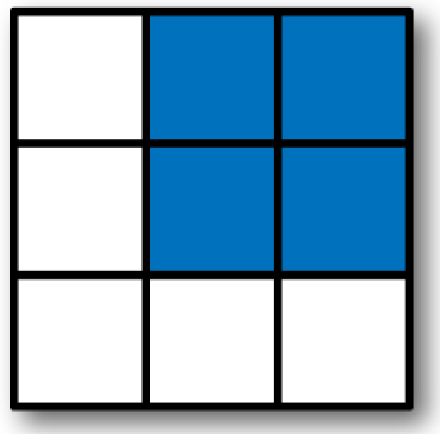


Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)
arr[:, :2]	(3, 2)

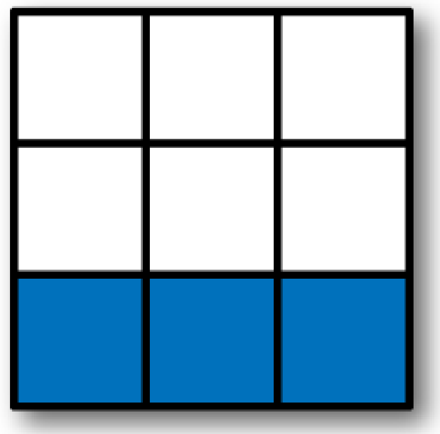
[W. McKinney, Python for Data Analysis]

2D Array Slicing

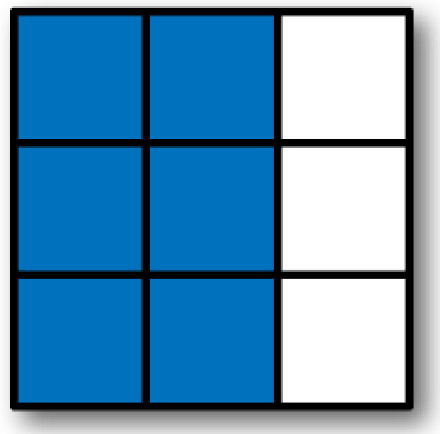
How to obtain the blue slice from array `arr`?



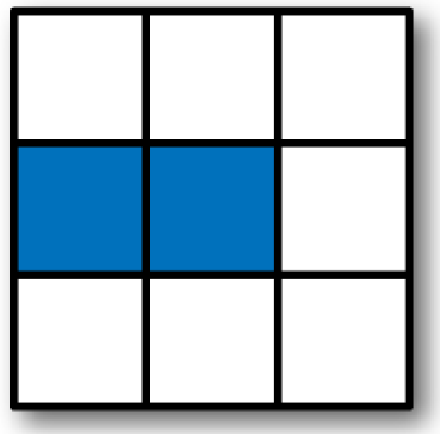
Expression	Shape
`arr[:2, 1:]`	(2, 2)



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]

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**

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`

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]) # ???`

Unary Universal Functions

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values
sqrt	Compute the square root of each element (equivalent to arr ** 0.5)
square	Compute the square of each element (equivalent to arr ** 2)
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
ceil	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as a separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise (equivalent to ~arr).

[W. McKinney, Python for Data Analysis]

Binary Universal Functions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

[W. McKinney, Python for Data Analysis]

Statistical Methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; zero-length arrays have `NaN` mean
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`)
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

[W. McKinney, Python for Data Analysis]

More

- Other methods:
  - `any` and `all`
  - `sort`
  - `unique`
- Linear Algebra (`numpy.linalg`)
- Pseudorandom Number Generation (`numpy.random`)