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**
Create Table

• An SQL relation is defined using the create table command:
  
  \[
  \text{create table } r (A_1 \, D_1, A_2 \, D_2, \ldots, A_n \, D_n, (C_1), \ldots, (C_k))
  \]

  - \( r \) 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:

  ```sql
  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
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
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`)
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
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.
Group By

- Find the average salary of instructors in each department

\[
\text{select dept\_name, } \text{avg(salary) as avg\_salary} \\
\text{from instructor} \\
\text{group by dept\_name;}
\]

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

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>
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');`
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.

\[
\text{- insert into instructor } \\
\quad \text{ select ID, name, dept_name, 18000 } \\
\quad \text{ from student } \\
\quad \text{ where dept_name = 'Music' and total_cred > 144;}
\]

- The select-from-where statement is evaluated fully before any of its results are inserted into the relation.

- If not queries like

\[
\text{insert into table1 select * from table1}
\]

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);
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
## Join Examples

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

### Left Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>

### Right Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Join Examples

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

(Full) Outer Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Inner Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>BIO-301</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>CS-190</td>
</tr>
</tbody>
</table>

[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

- 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)

<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>

Table 4-2. NumPy data types

- int8, uint8: Signed and unsigned 8-bit (1 byte) integer types
- int16, uint16: Signed and unsigned 16-bit integer types
- int32, uint32: Signed and unsigned 32-bit integer types
- int64, uint64: Signed and unsigned 64-bit integer types
- float16: Half-precision floating point
- float32: Standard single-precision floating point; compatible with C float
- float64: Standard double-precision floating point; compatible with C double and Python float object
- float128: Extended-precision floating point
- complex64, complex128, complex256: Complex numbers represented by two 32, 64, or 128 floats, respectively
- bool: Boolean type storing True and False values
- object: Python object type; a value can be any Python object
- string_: Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
- unicode_: Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')
**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 $\rightarrow$ 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

- \( a = \text{np.array}([1, 2, 3]) \)
  \( b = \text{np.array}([6, 4, 3]) \)

- (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`
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 $\text{arr3d}$:

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

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

```
old_values = $\text{arr3d}[0].\text{copy}()$
```

```
$\text{arr3d}[0] = 42$
```

```
$\text{arr3d}$
```

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

```
$\text{old_values}$
```

```
$\text{arr3d}$
```

[2D Indexing]
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`?

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

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]</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[:, 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>
</tbody>
</table>

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

How to obtain the blue slice from array \texttt{arr}?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{arr[2, :]}</td>
<td>(3,)</td>
</tr>
<tr>
<td>\texttt{arr[2]}</td>
<td>(3,)</td>
</tr>
<tr>
<td>\texttt{arr[:2, :]}</td>
<td>(1, 3)</td>
</tr>
<tr>
<td>\texttt{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}[::, 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>
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

<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&lt;sup&gt;x&lt;/sup&gt; 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, isnan</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, arccosh, 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>

---

D. Koop, CSCI 680/490, Spring 2022
### 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 $&gt;$, $\geq$, $&lt;$, $\leq$, $=$, $\neq$)</td>
</tr>
<tr>
<td>equal, not_equal</td>
<td>Compute element-wise truth value of logical operation (equivalent to infix operators $&amp;$, $</td>
</tr>
<tr>
<td>logical_and,</td>
<td></td>
</tr>
<tr>
<td>logical_or, logical_xor</td>
<td></td>
</tr>
</tbody>
</table>

[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:

```python
In [184]:
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```python
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:

```python
In [186]:
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```python
In [187]:
arr.cumsum()
```

```
Out [187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```python
In [188]:
arr.cumsum(axis=0)
```

```
Out [188]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```python
In [189]:
arr.cumprod(axis=1)
```

```
Out [189]:
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 4-5. Basic array statistical methods

<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>
More

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