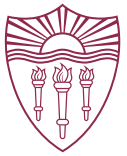
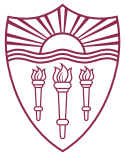


DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

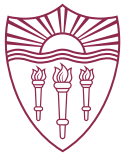
Itay Hen



JavaScript Object Notation

JSON

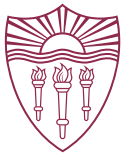
JSON: **J**ava**S**cript **O**bject **N**otation



- Lightweight data interchange format
- JSON data is structured as key-value pairs
- Keys are always strings enclosed in double quotes
- Values can be strings, numbers, objects {}, arrays [], booleans, null, or nested JSON objects

```
{
    "name": "John",
    "age": 30,
    "city": "New York",
    "birth_date":
    {
        "day": 14,
        "month": 4,
        "year": 1993
    },
    "skills":
    [
        "Python",
        "Machine Learning",
        "Databases"
    ],
    "awards": null,
    "male": true
}
```

JSON: Values Data Types



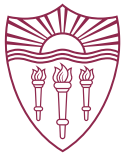
- **Strings**: Enclosed in double quotes.
- **Numbers**: Integer or floating-point values.
- **Objects**: Comma separated key-value pairs enclosed in curly braces.
- **Arrays**: Ordered lists of comma separated items enclosed in square brackets.
- **Booleans**: true or false.
- **null**: Represents empty or missing data.

JSON: Key Features

- **Simplicity**: JSON is easy for both humans to read and write and for machines to parse and generate
- **Interoperability**: It is supported by most programming languages
- **Data Exchange**: JSON is widely used for data exchange in web applications and APIs

https://www.json.org/json-en.html

XML vs JSON



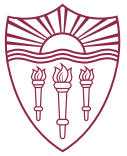
XML:

```
<person>
   <name> John </name>
   <phone> 303-4456 </phone>
</person>
```

JSON:

```
{"name": "John", "phone": "303-4456"}
```

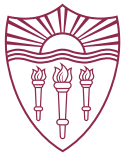
JSON in Python



- The json module is a built-in Python library that allows us to encode (serialize) and decode (deserialize) data using the JSON format.
- It's the standard way to exchange structured data between Python programs and web APIs, configuration files, etc.
- It has 4 main core functions:

Function	Direction	Description
`json.loads(str)`	JSON → Python	Parse a JSON string into a Python object
`json.load(file)`	JSON → Python	Parse JSON from a file
`json.dumps(obj)`	Python → JSON	Convert a Python object to a JSON string
`json.dump(obj, file)`	Python → JSON	Write JSON to a file

JSON in Python



JSON Type	Example	Python Type
object	`{"a": 1, "b": 2}`	`dict`
array	`[1, 2, 3]`	`list`
string	`"hello"`	`str`
number	`42`	`int` or `float`
true / false	`true` / `false`	`True` / `False`
null	`null`	`None`

json.loads()

- Purpose: Parse a JSON string into a Python object.
- Input: JSON string (e.g., '{"a": 1, "b": 2}')
- Output: Python object (usually dict or list)

```
import json

# JSON data as a string
data = '{"name": "Ariela", "age": 13, "subjects": ["Math", "Science"]}'

# Convert JSON string → Python object
person = json.loads(data)
```

What type is person?

json.loads()

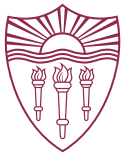
- Purpose: Parse a JSON string into a Python object.
- Input: JSON string (e.g., '{"a": 1, "b": 2}')
- Output: Python object (usually dict or list)

```python
import json

# JSON data as a string
data = '{"name": "Ariela", "age": 13, "subjects": ["Math", "Science"]}'

# Convert JSON string → Python object
person = json.loads(data)
```

dict:

```python
# Access elements like a normal Python dict
print(person["name"])       # Output: Ariela
print(person["age"])        # Output: 13
print(person["subjects"])   # Output: ['Math', 'Science']
```

json.load()



- Purpose: Parse JSON data directly from a file.
- Input: File object containing JSON text
- Output: Python object (dict, list, etc.)

```python
import json

# Open and read JSON data directly from a file
with open("data.json", "r") as f:
    person = json.load(f)

# Access values as a normal Python dict
print(person["name"])       # Output: Ariela
print(person["age"])        # Output: 13
print(person["subjects"])   # Output: ['Math', 'Science']
```

json.dumps()

- Purpose: Convert a Python object into a JSON string.
- Input: Python dict, list, etc.
- Output: JSON string
- Options:

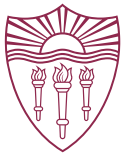
`indent`	Pretty-print	`indent=4`
`sort_keys`	Sort output keys	`sort_keys=True`
`separators`	Control punctuation spacing	`separators=(",", ":")`

```python
import json

# A normal Python dictionary
person = {
    "name": "Ariela",
    "age": 13,
    "subjects": ["Math", "Science"],
    "enrolled": True
}

# Convert to a JSON-formatted string
json_str = json.dumps(person)

print(json_str)
```

- Output:

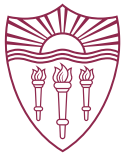
```
{"name": "Ariela", "age": 13, "subjects": ["Math", "Science"], "enrolled": true}
```

json.dump()



- Purpose: Write JSON representation of a Python object to a file.
- Input: Python object and open file handle
- Output: JSON text written to file

```python
import json

# A normal Python dictionary
person = {
    "name": "Ariela",
    "age": 13,
    "subjects": ["Math", "Science"],
    "enrolled": True
}

# Write the dictionary to a JSON file
with open("person.json", "w") as f:
    json.dump(person, f, indent=4)
```

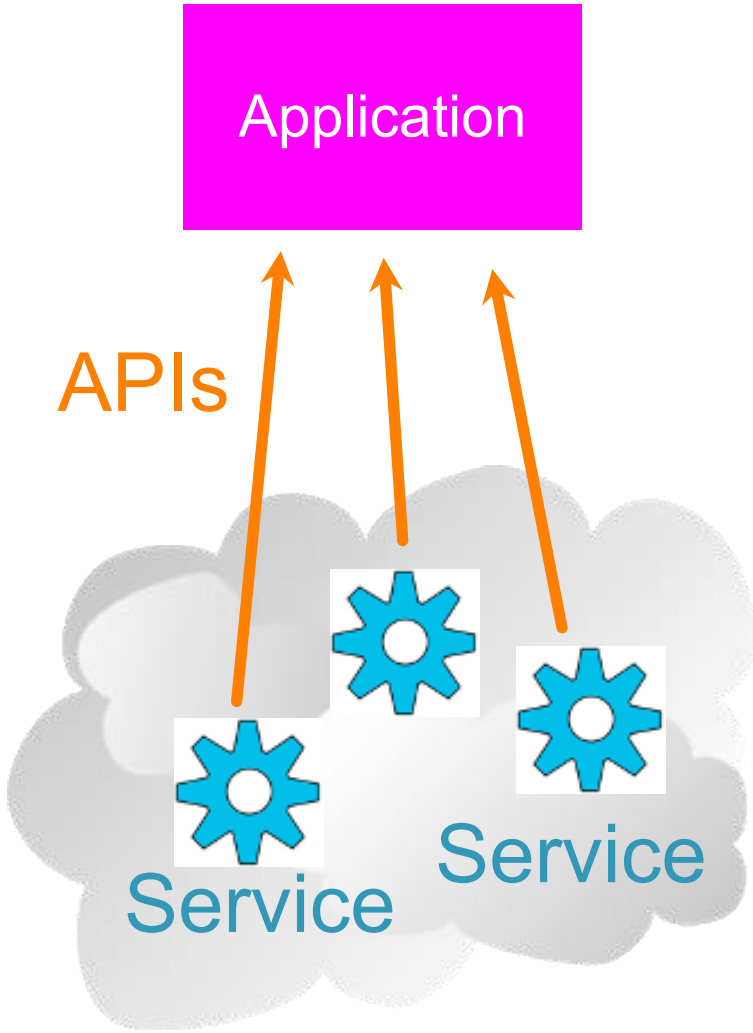


http://en.wikipedia.org/wiki/Service-oriented_architecture

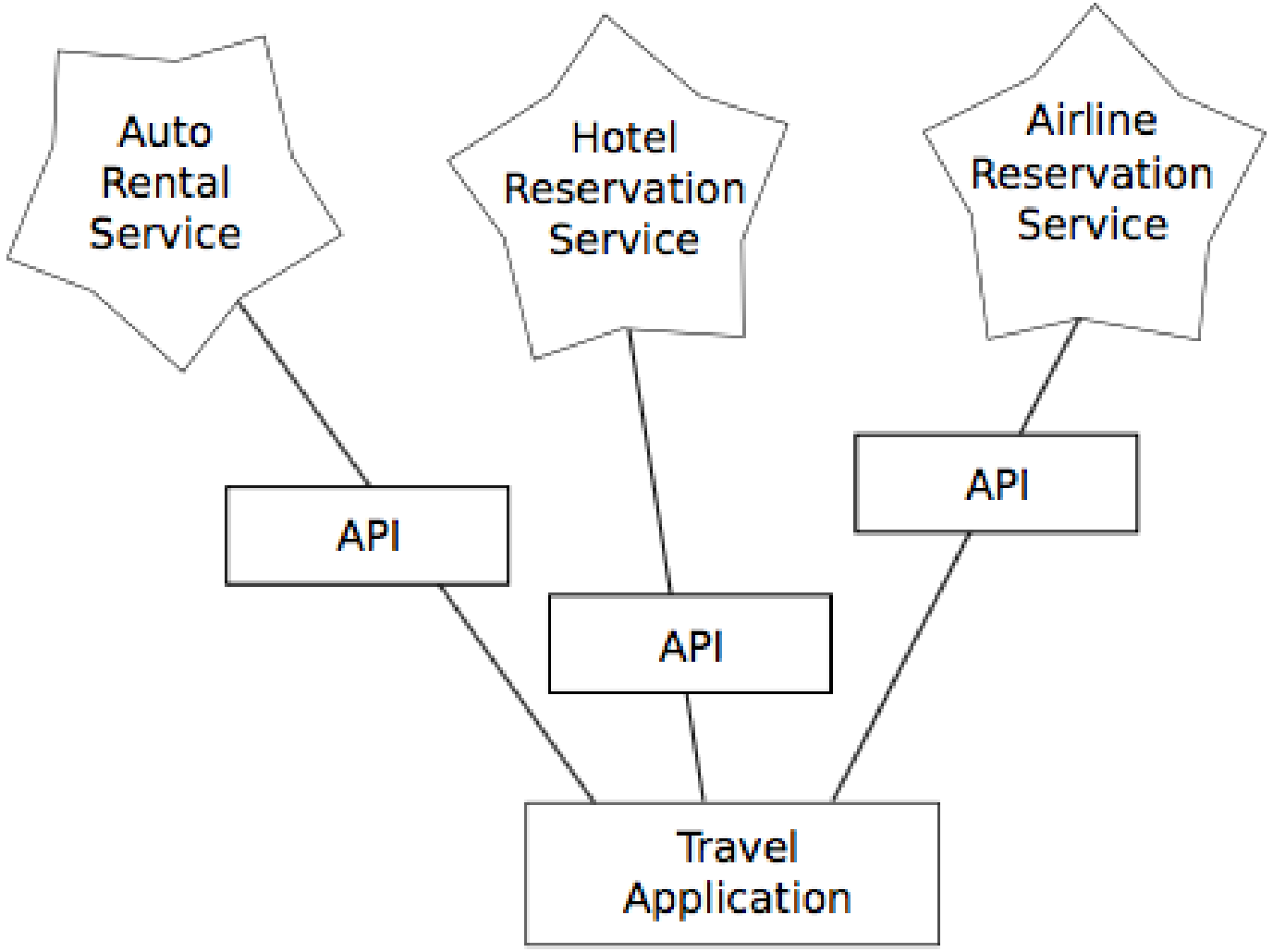
SERVICE ORIENTED APPROACH

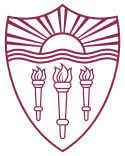
Service Oriented Approach

- Most non-trivial web applications use services
- They use services from other applications
  - Credit Card Charge
  - Hotel Reservation System
- Services publish the "rules", other applications must follow to make use of the service (API)



A Service Oriented Architecture

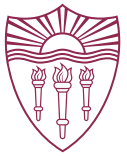




http://en.wikipedia.org/wiki/Web_services

WEB SERVICES

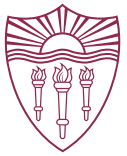
Application Program Interface (API)



- The API itself is largely abstract in that it specifies an interface and controls the behavior of the objects specified in that interface.
- The software that provides the functionality described by an API is said to be an “implementation” of the API.
- An API is typically defined in terms of the programming language used to build an application.

http://en.wikipedia.org/wiki/API

Application Program Interface (API)



- We interact with APIs every day:
  - Weather apps fetching forecasts from meteorological services
  - Payment processing with PayPal, or Square
  - Social media sharing buttons on websites
  - Google Maps integration in ride-sharing apps
  - Cloud storage services syncing your files
- Popular API services:
  - **Google Cloud APIs** – Maps, YouTube Data, Drive, Gmail, Translate, etc.
  - **Microsoft Graph API** – Integrates with Outlook, OneDrive, Teams, and more.
  - **Amazon Web Services (AWS)** – Provides programmatic access to cloud computing resources.
  - **Twitter (X) API** – Post, read, and analyze tweets.
  - **Facebook Graph API** – Access user posts, pages, and advertising data.
  - **Instagram Graph API** – For professional account insights and content.

Making GET Requests to an API service

GET requests retrieve data from an API endpoint.

```
import requests

response = requests.get('https://api.example.com/users')

# With parameters
params = {'page': 1, 'limit': 10}
response = requests.get('https://api.example.com/users', params=params)
```

Common use cases:

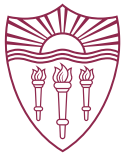
Fetching user data, retrieving lists

Example: Fetching Weather Data via an API

```python
import requests

# Define the base URL for the OpenWeatherMap API
base_url = "https://api.openweathermap.org/data/2.5/weather"

# Define query parameters
params = {
    "q": "Los Angeles",   # city name
    "appid": "YOUR_API_KEY_HERE",  # your API key
    "units": "metric"     # or 'imperial' for Fahrenheit
}

# Make the GET request with parameters
response = requests.get(base_url, params=params)
```

Very similar to fetching an HTML

Example: Fetching Weather Data via an API



1. **Base URL:** The API endpoint — here it's `https://api.openweathermap.org/data/2.5/weather`.
2. **Parameters (`params` dict):**
   - `q`: The city name.
   - `appid`: Your personal API key (you get this after registering on the site).
   - `units`: Optional parameter to control units.
3. `requests.get(url, params=params)` automatically encodes the parameters into a proper query string, e.g.

```
https://api.openweathermap.org/data/2.5/weather?q=Los%20Angeles&appid=YOUR_API_KEY_
```

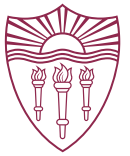
4. `response.json()` converts the JSON data returned by the API into a Python dictionary.

Example: Fetching Weather Data via an API

```
# Check if the request was successful (HTTP 200 OK)
if response.status_code == 200:
    data = response.json()  # Parse JSON response
    # Extract useful information
    city = data["name"]
    temp = data["main"]["temp"]
    weather = data["weather"][0]["description"]

    print(f"City: {city}")
    print(f"Temperature: {temp}°C")
    print(f"Weather: {weather}")
else:
    print(f"Error {response.status_code}: {response.text}")
```

We can do whatever we want with the data; our own analysis

Another example: retrieve data from YouTube



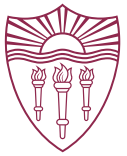
```python
import requests

# Replace with your actual API key
API_KEY = "YOUR_YOUTUBE_API_KEY"
query = "Quantum Computing"

url = "https://www.googleapis.com/youtube/v3/search"

# Passing parameters using the 'params' argument (cle
params = {
    "part": "snippet",
    "q": query,
    "type": "video",
    "maxResults": 5,
    "key": API_KEY
}
```

How do we know what parameters are legit?

- Look at the API documentation
- Visit the API's developer portal

```python
if response.status_code == 200:
    results = response.json()
    for item in results["items"]:
        title = item["snippet"]["title"]
        channel = item["snippet"]["channelTitle"]
        print(f"{title} – {channel}")
else:
    print("Error:", response.status_code)
```

Search queries/ query parameters

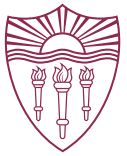


```python
import requests

url = "https://api.openweathermap.org/data/2.5/weather"
params = {
    "q": "Los Angeles",
    "units": "metric",
    "appid": "12345"
}

response = requests.get(url, params=params)
print(response.url)
```

Same as URL with a query string (often just “a query URL”):

```
https://api.openweathermap.org/data/2.5/weather?q=Los+Angeles&units=metric&appid=12345
```

API tokens/keys



An **API token** (or **API key**) is a **unique identifier** that lets an API server know *who you are* and what you're allowed to do.

It's essentially your **digital pass** to access an API — like a password, but used programmatically.

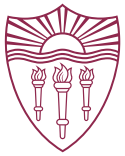
APIs issue tokens to:

1. **Authenticate** – Verify that the request is coming from an authorized user or app.
2. **Authorize** – Decide what that user/app is allowed to do (e.g. read-only vs. write access)
3. **Rate-limit** – Track how many requests you're making (to prevent abuse).
4. **Audit/log** – Record who did what and when.

Without a token, most APIs will reject your requests with something like:

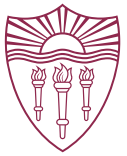
- Registering for an API token is often **free** for basic or educational use.
- Many services like **OpenWeatherMap**, **YouTube Data API**, and **NASA APIs** provide free tokens with limited request quotas.

APIs Beyond fetching data (beyond scope of class)



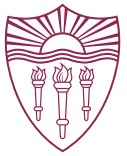
- 🔄 **Send or update data** — e.g., create new GitHub issues, post tweets, upload files to Google Drive, or push data to a database using `POST` or `PUT` requests.
- 🧩 **Automate workflows** — chain multiple APIs together (e.g., get weather → update smart lights → send a Slack alert).
- 🤖 **Control hardware or devices** — trigger IoT devices, smart thermostats, or sensors using REST or MQTT APIs.
- 💼 **Integrate services** — connect systems like Stripe (payments), Twilio (SMS), and Notion (notes) into your own app or backend.
- 📬 **Send notifications** — use APIs to send emails, text messages, or push notifications automatically.
- 🧠 **Use AI and analytics** — call APIs for language models, image recognition, or sentiment analysis (like OpenAI's GPT or Google Vision).
- 🧾 **Manage user accounts** — authenticate users, manage sessions, or access OAuth-protected resources (Google, Facebook, GitHub logins).

RESTful APIS: Key concepts

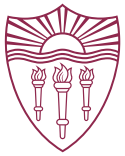


- **REST** stands for **Representational State Transfer** — an architectural style for building web services.
- A **RESTful API** follows REST principles to allow clients and servers to communicate over HTTP in a simple, stateless way.
- It uses **standard HTTP methods** — `GET`, `POST`, `PUT`, `DELETE`, (and sometimes `PATCH`, `HEAD`, `OPTIONS`) — each with a specific semantic meaning.
- **Resources** are the core concept — everything (users, files, posts, etc.) is treated as a *resource* identified by a unique **URL** (endpoint).
  - Example: `/users`, `/users/42`, `/posts/17/comments`
- Communication is **stateless** — each request from the client contains all necessary information; the server doesn't remember previous interactions.
- **Uniform interface:** the same structure and conventions are used for all resources (e.g., `/resource/id`).
- **Data formats:** JSON is most common, but XML, YAML, or plain text can also be used.

RESTful APIS: Key concepts

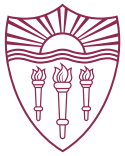


- **Query parameters** (after `?`) can refine or filter data, e.g. `/users?role=admin&active=true`.
- **HTTP status codes** indicate results (e.g. `200 OK`, `201 Created`, `404 Not Found`, `500 Internal Server Error`).
- **REST APIs are stateless and cacheable**, improving scalability and performance.
- They are **language-agnostic** — any client (Python, JavaScript, C++, etc.) can interact with them as long as it can send HTTP requests.
- The **server** exposes endpoints, and the **client** consumes them using HTTP requests and responses.
- Example REST interaction:
  - `GET /users/42` → retrieve user data
  - `PUT /users/42` → update user data
  - `DELETE /users/42` → remove the user



DATA ANALYSIS

What is Data Analysis?



- The process of inspecting, cleaning, transforming and modeling data to discover useful information and make informed decisions.
- Key Steps
  - Data Collection
  - Data Cleaning
  - Data Analysis and Exploration
  - Data Visualization

https://en.wikipedia.org/wiki/Data_analysis

Data Analysis Process

- SQL
- Scrapping
- File Formats
  - CSV
  - JSON
  - XML
- Consulting APIs
- Buying Data
- Distributed Databases

- Missing values and empty data
- Data imputation
- Incorrect types
- Incorrect or invalid values
- Outliers and non relevant data
- Statistical sanitization

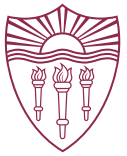
- Hierarchical Data
- Handling categorical data
- Reshaping and transforming structures
- Indexing data for quick access
- Merging, combining and joining data

- Exploration
- Building statistical models
- Visualization and representations
- Correlation vs Causation analysis
- Hypothesis testing
- Statistical analysis
- Reporting

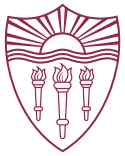
- Building Machine Learning Models
- Feature Engineering
- Moving ML into production
- Building ETL pipelines
- Live dashboard and reporting
- Decision making and real-life tests

RMOTR BY INE

Python Libraries for Data Analysis



- pandas: tabular data manipulation
- numpy: numeric library for n-dimensional arrays, many mathematical functions: linear algebra, random numbers
- matplotlib: data visualization, many plots.
- seaborn: statistical visualization, built on top of matplotlib
- statsmodels: advanced statistical functions
- scipy: advanced scientific computing, including functions for optimization, linear algebra, differential equations, integration
- scikit-learn: most popular machine learning library for Python (not DL)
- TensorFlow, PyTorch: deep learning



pip install numpy

NUMPY

What Is NumPy?

- **NumPy** = **Numerical Python** — the fundamental package for scientific computing in Python.
- Provides powerful **multi-dimensional arrays** (`ndarray`).
- Enables **fast, vectorized** operations (written in C under the hood).
- Core of the **Python scientific stack** — used by SciPy, Pandas, TensorFlow, and more.

Why use NumPy?

- Much **faster** than native Python lists for numerical operations.
- Offers **element-wise arithmetic** and broadcasting.
- Has built-in **linear algebra**, **FFT**, and **random number** capabilities.
- Seamless **integration with C/C++ and Fortran code**.
- Key for **data analysis**, **machine learning**, and **simulation** workflows.

Core Data Structure: ndarray

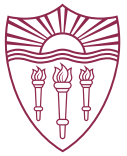
- Represents a **homogeneous, n-dimensional** array of fixed-type elements.
- Example:

```
import numpy as np
a = np.array([[1, 2, 3],
              [4, 5, 6]])
```

What kind of array is this? What are its dimensions?

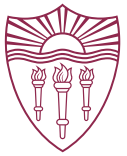
- Attributes:
  - `a.shape` → dimensions (2, 3)
  - `a.ndim` → number of axes (2)
  - `a.dtype` → element type (int, float, etc.)
  - `a.size` → total number of elements

ndarray attributes



- .dtype: an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally, NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- .itemsize: the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to.dtype.itemsize.

ndarray **caveats**



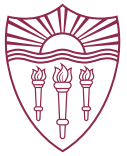
- **Watch out for this error:**

```
>>> a = np.array(1, 2, 3, 4)    # WRONG
>>> a = np.array([1, 2, 3, 4])  # RIGHT
```

- array **transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.**

```
>>> b = np.array([(1.5, 2, 3), (4, 5, 6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

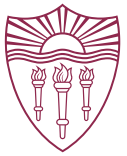
ndarray **caveats**



Often, the elements of an array are originally unknown, but its size is known.
NumPy makes creating arrays with placeholder content easy.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones((2,3,4), dtype=np.int16) # can also specify dtype
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
```

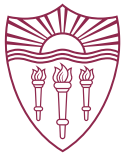
ndarray **caveats**



- arrays can be generated with .arrange, similar to using range to get a list.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
# it accepts float arguments
>>> np.arange( 0, 2, 0.3 )
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

Vectorization & Broadcasting



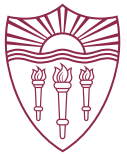
- NumPy eliminates explicit Python loops by performing **vectorized operations**:

```
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])
z = x + y       # → array([11, 22, 33])
```

- Broadcasting automatically expands smaller arrays to match shapes:

```
x = np.array([1, 2, 3])
y = 10
x + y           # → array([11, 12, 13])
```

Common functions



- Array creation

```
np.zeros((3,3)), np.ones(5), np.arange(0,10,2), np.linspace(0,1,5)
```

- Statistics

```
a.mean(), a.std(), a.sum(), a.min(), a.max()
```

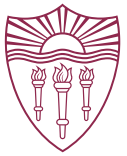
- Linear algebra

```
np.dot(A, B), np.linalg.inv(A), np.linalg.eig(A)
```

- Random

```
np.random.rand(3,3), np.random.normal(0,1,100)
```

Example

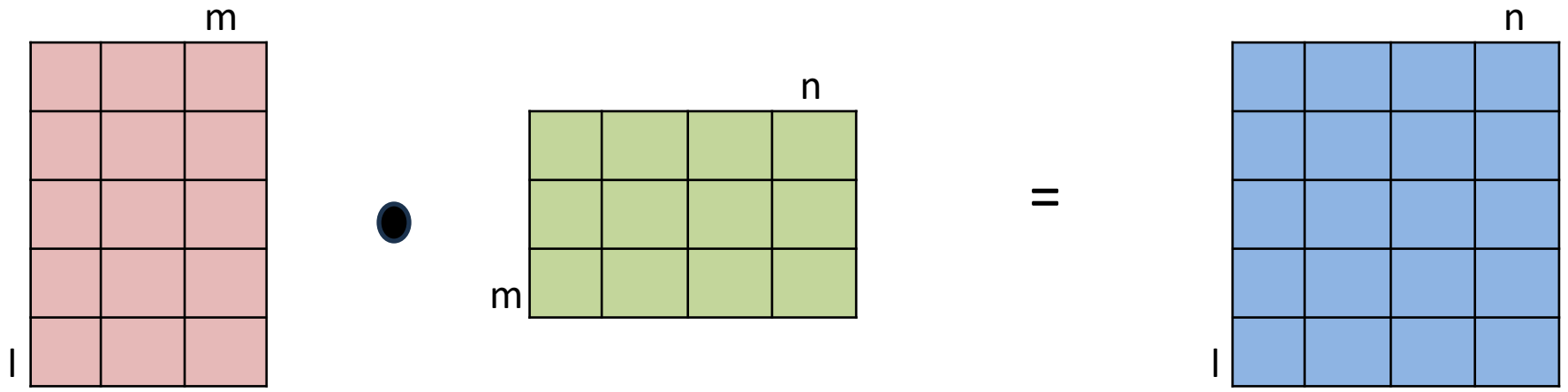


```python
import numpy as np


# Create 1000 random points and compute distances
x = np.random.rand(1000)
y = np.random.rand(1000)

dist = np.sqrt(x**2 + y**2)
print(np.mean(dist))
```

creates a 1-D array of length 1000 with values uniformly sampled from **[0, 1)**.

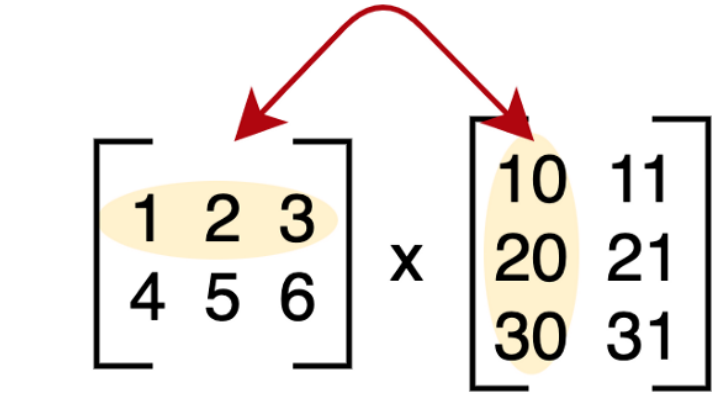
For each sample: $\text{dist}_i = \sqrt{x_i^2 + y_i^2}$

Matrix Multiplication



A • B = C

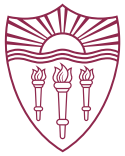
Matrix Multiplication Example



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

$$= \begin{bmatrix} 1x10 + 2x20 + 3x30 & 1x11 + 2x21 + 3x31 \\ 4x10 + 5x20 + 6x30 & 4x11 + 5x21 + 6x31 \end{bmatrix}$$

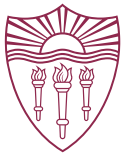
$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$



- import numpy as np

- matrix1 = np.array([[1, 2], [3, 4]])
- matrix2 = np.array([[5, 6], [7, 8]]
- result = np.dot(matrix1, matrix2)
- The number of columns in the first matrix must equal the number of rows in the second matrix.
- Resulting matrix dimensions: (rows1, columns2)

Matrix Multiplication Applications



- Computer Graphics (Video Games, Video Editing)
- Robotics
- Artificial Intelligence (Machine Learning, Deep Learning)
- Linear algebra
- Network theory
- Bitcoin mining
- …and many more