Venn Diagram in Python
Nathan Crowley - 118429092

What are Venn Diagrams:

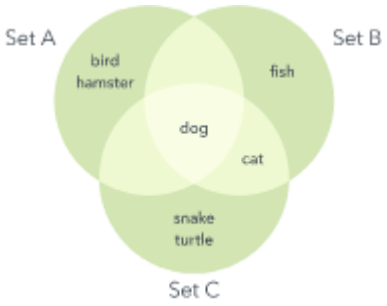
- Diagram style that shows the logical relationship between sets by John Venn 1880's.
- Illustrates simple set relationships.
- Venn Diagram uses simple closed curves drawn on a plane to represent sets, very often circles or ellipses.
- Overlapping circles or other shapes to illustrate the logical relationships between two or more sets of items.
- Allow users to visualise data in a clear illustration.

Venn Diagram history:

- John Venn wrote a paper in 1880 entitled "On the diagrammatic and Mechanical Representation of Propositions and Reasonings"
- But roots of this type of diagram date back to the 1200s, philosopher and logician Ramon Llull of Majorca used a similar type of diagram.

Venn Diagram Examples:

- Set A: dog, bird, hamster
- Set B: dog, cat, fish
- Set C: dog, cat, turtle, snake
    - Overlap / intersection = dog



https://www.lucidchart.com/pages/tutorial/venn-diagram

Advantages and Disadvantages

Advantages:

- Visually organize information in regards to Intersections, Unions,etc.
- Comparing two sets of information and presenting clearly the shared information and how they differ.

Disadvantages:

- Can be difficult to view with more than three sets.

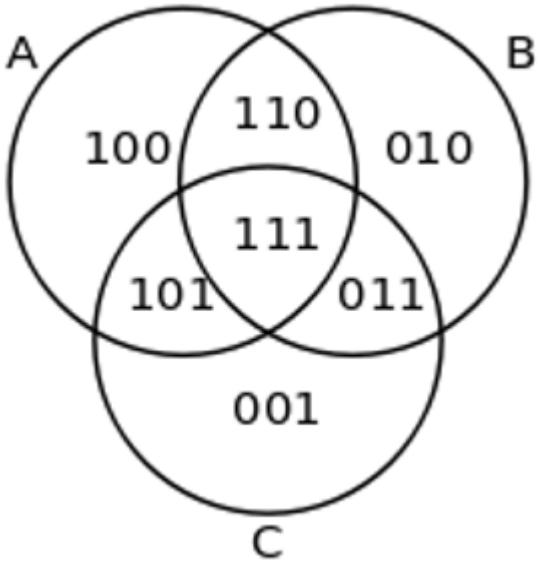
Create Venn Diagrams in Python

1) **Install Libraries and import**

2) **Venn diagram with 2 sets**

3) **Customizing the Venn Diagram**

4) **Venn diagram with 3 sets**



Source: https://en.wikipedia.org/wiki/Venn_diagram

5) **Venn diagram with 6 sets**

21/11/21

What is a Python Library?

Python Library is a reusable chunk of code that you may want to include in your programs/projects.
Developers can create libraries with a wide range of functionality, allowing users/other developers to import their libraries and take advantage of the already created functionality.

Create Python Library (example):

1) Create a directory in which you want to put your library

Inside Year_4/FinalYearProject create an inner directory to contain the Python Library Example.
In this example the directory name: **mypythonlibrary**.

2) Create a virtual environment for your folder

Virtual Environments are used to install packages locally to the current working directory, instead of installing globally, which could cause system issues OR future updates of the globally installed packages may cause issues with the functionality of the current working directory.
Activating the virtual environment modifies the PATH environment variable to point to the specific isolated Python set-up you created.

A virtual environment = environment containing Python along with necessary libraries and scripts for the functionality of the environment.

- **Create virtual environment**

  ```
  > python3 -m venv [name]
  ```

- **Activate the virtual environment**

  ```
  > source [name]/bin/activate
  ```

Once the VE is created and activated, install the packages needed for the functionality of this VE.

- **Install packages needed**

  > pip install wheel
  > pip install setuptools
  > pip install twine

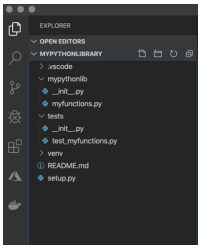
3) Create a folder structure

Any folder that has an __init__.py will be included in the library when we build it. Upon import, the code inside each __init__ gets executed, so should be minimal code just what's needed to run your project.

Structure:----------

a) **setup.py** - create an empty file - MOST IMPORTANT!
b) **README.md** - describe the contents of the library to others.
c) **mypythonlib OR [name]** - folder that people will use to pip install your lib
d) **cd mypythonlib OR [name]**
   i) **__init__.py** - file inside *mypythonlib OR [name].*
   ii) **myfunctions.py OR [name].py** - folder inside *mypythonlib OR [name].*
e) **test folder** - inside root.
   i) **__init__,py** - inside the test folder.
   ii) **test_myfunctions.py** - inside the test folder

NOTE:**********
> cd mypythonlib
inside __init__.py:
> from mypythonlib import myfunctions
****



4) Create content for your library

When creating functions for your library, put them in the **myfunctions.py** file.
Whenever you write any code, write **tests** for this code.

Write Code:----------

For this example - copy the *haversine function* into the file:
This function will give us the distance in meters between two latitude and longitude points.

```
from math import radians, cos, sin, asin, sqrt
def haversine(lon1: float, lat1: float, lon2: float, lat2: float) -> float:
    """
    Calculate the great circle distance between two points on the
    earth (specified in decimal degrees), returns the distance in
    meters.
    All arguments must be of equal length.
    :param lon1: longitude of first place
    :param lat1: latitude of first place
    :param lon2: longitude of second place
    :param lat2: latitude of second place
    :return: distance in meters between the two sets of coordinates
    """
    # Convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers
    return c * r
```

Testing:----------

Whenever you write any code, write tests for this code - **pytest** and **pytest-runner**.
Install the libraries in the virtual environment.

```
> pip install pytest==4.4.1
> pip install pytest-runner=4.4
```

Let's create a small test for the *haversine function*, inside the **test/test_myfucntions.py**:

```
from mypythonlib import myfunctions
def test_haversine():
   assert myfunctions.haversine(52.370216, 4.895168, 52.520008,
   13.404954) == 945793.4375088713
```

Create setup.py:----------

Setup.py file helps us build the library.
Limited example of setup.py:

```
from setuptools import find_packages, setup

VERSION = '0.0.1'
DESCRIPTION = "python library example for learning how to create libs"

setup(
   name='mypythonlib',
  version=VERSION,
  author="Nathan Crowley",
  author_email="<crowley2305@gmail.com>",
  description=DESCRIPTION,
   packages=find_packages(),
   install_requirements=[ ],
   license='MIT',
   classifiers=[
        "Development Status :: 1 - Planning",
        "Intended Audience :: Developers",
        "Programming Language :: Python :: 3",
        "Operating System :: Unix",
        "Operating System :: MacOS :: MacOS X",
        "Operating System :: Microsoft :: Windows",
  ]
)
```

name variable - holds whatever name you want your package wheel file to have, to make it easy use the same name as the folder.

a) Set the packages you would like to create

Note that **pip** does not use **requirements.yml / requirements.txt** when your project is installed as a dependency by users. For this, you should specify dependencies in the **setup.py** file:

```
setup(
        ……
        install_requires ….
        test_require ….
)
```

Install_requires - should be limited to the list of packages that are ***absolutely needed***. However since we are only using the *math library* (which is always available in Python), we can leave it empty.

b) Set the requirements your library needs

****

Note: pytest doesnt need to be installed by users, but in order to have it installed *automatically only when you run test you can add to* ***setup.py****:*

```
from setuptools import find_packages, setup

setup(
    name='mypythonlib',
    packages=find_packages(include=['mypythonlib']),
    version='0.1.0',
    description='My first Python library',
    author='Me',
    license='MIT',
    install_requires=[],
    setup_requires=['pytest-runner'],
    tests_require=['pytest==4.4.1'],
    test_suite='tests',
)
```

Running:

```
> python setup.py pytest
```

Will execute all tests stored in **test** folder.

****

5) Build your library

Once the content for the functionality is created, we need to build the library to have it published (PyPI repository) and installed from there.

- Ensure current working directory = *path/to/mypythonlibrary* (same directory as setup.py and README.md)
- Run:

  ```
  > python setup.py bdist_wheel
  ```
- Wheel file stored in **'dist' folder** that is created.
- Isntall library using:

  ```
  > pip install /path/to/wheelfile.whl
  ```

a) Upload to PyPI

- You can publish the library to PyPI repo and install it from there.
- You can publish anything to PyPI no filter for good vs bad packages.
  1) Create an account on PyPI
  2) `> twine upload dist/*`
  3) Fill in username and password from PyPI account
- Once uploaded people can install with:

  ```
  > pip3 install [setup.py name attribute]
  ```

- Once you install the library, you can import by:
  > import mypythonlib
  > from mypythonlib import myfunctions

20/10/21

What is an API?

- "API allows for information or functionality to be manipulated by other programs via the internet."
- Application Programming Interface = part of a computer program designed to be used or manipulated by another **program**.
- (As opposed to being used by another human)

When to create an API?

1) Large dataset.
2) Data to be accessed in real-time.
3) Data will be changed/be updated frequently.
4) Users (other programs) only access data one part at a time.
5) Advanced actions to be performed other than retrieve data, such as:
   a) Contributing
   b) Updating data
   c) Deleting data.

- If you have data to share to the world of developers, use an API.
- If data set size relatively small just provide a 'data dump', but can be beneficial to provide API and data dump.

Data dump = size of data relatively small, instead provide a 'data dump' in form of a downloadable JSON/ XML/ CSV/ SQLite file.

API Terminology:

1) **HTTP:**
   - HyperText Transfer Protocol, primary means of **data communication on the web**.
   - **'Methods'** used to show data direction: ['GET', 'POST',....]

2) **URL ("`https://programminghistorian.org/about`")**
   - **Address** for a resource on the web.
   - Consists of:
     1) Protocol - `https://`
     2) Path - `programminghistorian.org/about`

3) **JSON:**
   - JavaScript Object Notation, text-based data storage format.
   - Designed to be easy to read for humans **and** machines.
   - Most common **format for returning data** through API.

4) **REST**
   - REpresenational State Transfer, describes best practices for implementing API's.
   - API designed with some or all rules are called RESTful API.
   - Can be disagreement around the principles needed, so may refer tomas web API or HTTP API.

Endpoint = a URL that enables the API to access resources on a server, often through a RESTful API interface.

Using API's

- Case study - Sensationalism and Historical Fires

  Our research area is sensationalism and the press: has newspaper coverage of major events in the US become more or less sensational over time? Narrowing the topic, we may ask whether coverage of *urban fires* has increased or decreased with government reporting on fire-related relief spending?

  Begin by collecting historical data on newspaper coverage of fires using an API - http://chroniclingamerica.loc.gov/about/api/. This API does not require an authentication process.

  Our initial goal is to find *all newspaper stories in the database that use the term 'fire'*.

  Typically any use of an API starts with its documentation, we find two critical piece of information:

  1) API's **Base URL** `http://chroniclingamerica.loc.gov`
  2) **Path** corresponding to functions you want to preform (search) `/search/pages/results/`

3) If we **combine the 'Base URL' and the 'path' into one URL**, we will create a request to the API that returns all available data in the DB. http://chroniclingamerica.loc.gov/search/pages/results/

This will return data on all items available at the time of writing in a *formatted HTML view*. If we want structured data (JSON) and related to fires, we need to pass **Query Parameters** (?):

1) ***format=json*** - changes returning data from HTML to JSON.
2) ***proxtext=fire*** - narrows returned entries to those with search term 'fire'.

http://chroniclingamerica.loc.gov/search/pages/results/?format=json&proxtext=fire

- Query Parameters follow the "**?**" in the URL, and separated by "**&**".
- Example output structure:


```
"city": ["Washington"], "date": "19220611",
"title": "The Washington herald. [volume]",
"end_year": 1939, "note": ["Also issued on
microfilm from the Library of Congress,
Photoduplication Service.", "Archived issues
are available in digital format as part of the
Library of Congress Chronicling America online
collection.", "Nov. 19-20 both called vol 1,
no. 1", "On Sunday published as: Washington
times-herald, Nov. 19, 1922-Apr. 15, 1923;
Washington herald times, Sept. 26, 1937-Jan.
29, 1939."], "state": ["District of
Columbia"], "section_label": "", "type":
"page", "place_of_publication": "Washington,
D.C.", "start_year": 1906, "edition_label":
"Sunday Edition", "publisher": null,
"language": ["English"], "alt_title":
["Washington herald times", "Washington
times-herald"], "lccn": "sn83045433",
"country": "District of Columbia", "ocr_eng":
"Mrs. Harding with representative# of the Camp
Fire Girto.", "batch": "dlc_greyhound_ver02",
"title_normal": "washington herald.", "url":
"https://chroniclingamerica.loc.gov/lccn/sn830
45433/1922-06-11/ed-1/seq-37.json", "place":
["District of Columbia--Washington"], "page":
"Page 5"}, {"sequence": 32, "county": [null],
"edition": null, "frequency": "Daily", "id":
"/lccn/sn83045433/1922-07-30/ed-1/seq-32/",
"subject": ["Washington
(D.C.)--fast--(OCoLC)fst01204505", "Washington
(D.C.)--Newspapers."],
```

If we were to pursue this research further, a next step might be cleaning the data to reduce the number of false positives, or find how many newspapers about fire appear on the front page over time.

What users want from an API:

1) **Documentation** is a user's starting point when working with a new API.
2) **Well designed URLs** make it serier for users to intuitively find resources/endpoints.

Implementing our API

1. Overview
    - Build prototype API using **Python** and **Flask web framework.**
    - Our example API will take the form of a **distant reading archive** - *a book catalog that goes beyond standard bibliographic information to include data of interest to those working on digital projects. Our API will serve the **title, date of publication, and first sentence of each book**.*

2. Create basic Flask app
    - Why Flask?:
        Python has a number of web frameworks to create web apps and APIs (Django). Flask applications tend to be written on a **blank canvas**, and are more suited to a contained application such as an API.
    - Create a new folder on the computer that will be a project folder.
    - Create a file called **api.py** in a **api folder**:

```
import flask
app = flask.Flask(__name__)
app.config["DEBUG"] = True
@app.route('/', methods=['GET'])
def home():
    return "<h1>Distant Reading
Archive</h1><p>This site is a prototype API for
distant reading of science fiction novels.</p>"
app.run()
```

3. Running the app
    - Navigate to the **api folder:**

```
cd
~/FinalYearProject/API_research/API_prototype/api
```

    - Run the Flask application with command:

```
python api.py   #activate the virtual env
```

    - Expected output:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

4. What Flask does
    - Flask maps HTTP requests to Python functions.
        - In our case we mapped the **URL path "/"** to the function **"home"**.
    - Process of mapping URLs to functions = **routing**.

```
@app.route('/', methods=['GET'])
```

- 'GET' = get data from application
- 'POST' = send data from user.
- **Component explanation - Run a Flask application:**
  - `import flask`
    - Import the Flask Library.
  - `app = flask.Flask(__name__)`
    - Create Flask app object, which contains data about the object and methods (object functions) that tell the app to do certain actions. (`app.run()` is one such method)
  - `app.config["DEBUG"] = True`
    - Start the debugger. If the ocde malformed, you'll see an error when visiting the app.
  - `app.run()`
    - Runs the application server.

5. Creating the API
   - Add our **data as a list of Python dictionaries**.
   - Key = type of information represented.
   - Value = the actual data.

```
[{
        'name': 'Alexander Graham Bell',
        'number': '1-333-444-5555'
    },
    {
        'name': 'Thomas A. Watson',
        'number': '1-444-555-6666'
    }
]
```

   - Our example is a Phone Book with each dictionary a phone book entry consisting of 'name' and 'number'.

   - For our Flask API, we add data on three science fiction novels (id, title, author, first_sentence, year_published).
   - `import flask`
   - `from flask import request, jsonify`

   - `app = flask.Flask(__name__)`
   - `app.config["DEBUG"] = True`

   - `# Create some test data for our catalog in the form of a list of dictionaries.`
   - `books = [`
   - `    {'id': 0,`
   - `     'title': 'A Fire Upon the Deep',`
   - `     'author': 'Vernor Vinge',`

```
    'first_sentence': 'The coldsleep itself was dreamless.',
    'year_published': '1992'},
   {'id': 1,
    'title': 'The Ones Who Walk Away From Omelas',
    'author': 'Ursula K. Le Guin',
    'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer came to the city Omelas, bright-towered by the sea.',
    'published': '1973'},
   {'id': 2,
    'title': 'Dhalgren',
    'author': 'Samuel R. Delany',
    'first_sentence': 'to wound the autumnal city.',
    'published': '1975'}
]

@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''

# A route to return all of the available entries in our catalog.
@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)
app.run()
```

6. Finding Specific Resources
   - Add functionality that allows users to **filter their returned data** using a more specific URL.
   - Add following code:

```
@app.route('/api/v1/resources/books', methods=['GET'])
def api_id():
    # Check if an ID was provided as part of the URL.
    # If ID is provided, assign it to a variable.
    # If no ID is provided, display an error in the browser.
    if 'id' in request.args:
        id = int(request.args['id'])
    else:
```

```
        return "Error: No id field provided. Please specify an id."

    # Create an empty list for our results
    results = []

    # Loop through the data and match results that fit the requested ID.
    # IDs are unique, but other fields might return many results
    for book in books:
        if book['id'] == id:
            results.append(book)

    # Use the jsonify function from Flask to convert our list of
    # Python dictionaries to the JSON format.
    return jsonify(results)
```

7. Understanding our Updated API
    - Created new function **api_id()**
    - **@app.route** used to map function to the path **/api/v1/resources/books**
    - Examine the provided URL and check the **query parameters** (?id=0)
    - Check the query parameter with **Python request.args**

API Design Principles

- Our next version will pull data from a database before providing it to a user.
- Also will take additional query parameters allowing the user to filter by other fields.

1) Designing Requests

- Modern APIs philosophy is **REST**:
    - Based on four methods of **HTTP protocol** -> actions performed on data in a **database**:

1) POST	->	CREATE
2) GET	->	READ
3) PUT	->	UPDATE
4) DELETE	->	DELETE

- How requests should be formatted:
    - Consider a poorly designed example of API endpoint:

`http://api.example.com/getbook/10`

        - First problem is **semantic**, in REST API our verbs are GET,POST,PUT,DELETE, nad are defined in the **Python request method** rather than the request URL.
        - The word 'get' *should not* appear in the request URL.
    - Incorporating these principles:
        http://api.example.com/books/10
        - The above request uses part of the path, **/10**, to provide the ID. This is somewhat inflexible - generally only filtered by one field at a time.
        - **Query Parameters (?id=10)** allow to filter multiple database fields.
        http://api.example.com/books?author=Ursula+K.+Le Guin&published=1969&output=xml
    - When designing how requests to your API should be structured, makes sense to **plan for future additions**. Even if the current API only serves one type of **resource**, you might add more resources in future, add **/resources/** to URL.
        http://api.example.com/resources/books?id=10
    - Another way to plan for the future is to add **version number** to the path **/v1/**.
    https://api.example.com/v1/resources/books?id=10

2) **Documentation and Examples**
    - Without documentation, even the best-designed API will be unusable.
    - Your API should have:
        - *Describing the resources.*
            - *Section for each resource that describes with fields it accepts (id, title), in form of a sample HTTP request or block of code.*
        - *Describing the functionality available.*
        - *Concrete working examples of request URLs or code.*
    - Common practice in documenting APIs is to provide **annotations** in your code that are automatically collated into documentation using a tool such as **Doxygen** or **Sphinx**.
        - Tools create documentation from **docstrings** - comments you make on your function definitions.
        - Helpful but not the end of the documentation.
    - Put yourself in the position of a potential user of your API and provide working examples.
    - For inspiration on how to approach API documentation:
        http://api.repo.nypl.org/

3) **Connecting API to Database**
    - This example of our API, pulls data from a database, implements error handling, and can filter books by publication date.
    - Database used is **SQLite**.
        download the example database
    - Copy the file to the **api** folder.
    - API code - **api_final.py**:

```
import flask
from flask import request, jsonify
import sqlite3
```

```
app = flask.Flask(__name__)
app.config["DEBUG"] = True

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d


@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science
fiction novels.</p>'''


@app.route('/api/v1/resources/books/all',
methods=['GET'])
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM
books;').fetchall()

    return jsonify(all_books)




@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not
be found.</p>", 404


@app.route('/api/v1/resources/books',
methods=['GET'])
def api_filter():
    query_parameters = request.args

    id = query_parameters.get('id')
    published = query_parameters.get('published')
    author = query_parameters.get('author')

    query = "SELECT * FROM books WHERE"
```

```
    to_filter = []

    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
    if not (id or published or author):
        return page_not_found(404)

    query = query[:-4] + ';'

    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()

    results = cur.execute(query,
to_filter).fetchall()

    return jsonify(results)

app.run()
```

- Run with **python api_final.py**.

4) Understanding Database-powered API

- Relational databases allow for storage and retrieval of data, stored in tables.
- Tables are similar to spreadsheets in that they have **columns and rows**.
    - **Columns = what the data represents** (title, date)
    - **Rows = individual entries** (books, users, any kind of entity)
- Our Database has five columns [id pulbished author title first_sentence]
- Each row represents one book that won the Hugo award in the year under the [published] column.
- Our **api_all** function pulls data from the Hugo database.

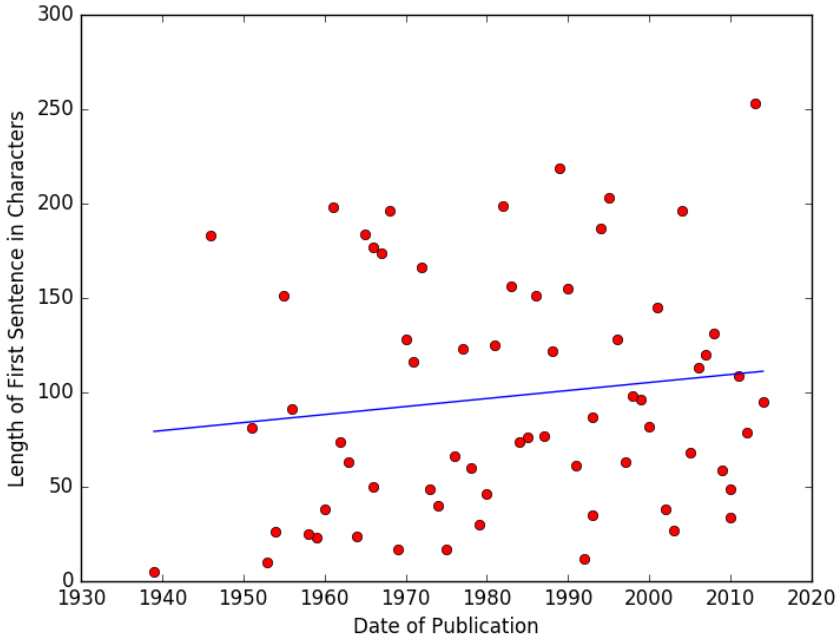
```
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM
books;').fetchall()

    return jsonify(all_books)
```

    - Connect to DB using sqlite3 library.
    - **conn** = object representing the connection to the database.

- **conn.row_factory = dict_factory** = lets connection object know to use the **dict_factory** function we defined earlier. Which returns items from the DB as dictionaries rather than lists - these better for JSON output.
- **cur = conn.cursor()** = create cursor object, which is what moves through the database to pull our data.
- cur.execute = execute an SQL query, to pull out all available data **(*)** from the **books table.**
- **jsonfiy(all_books)** = data returned is JSON.

5) API in practice



Resources:

- API research https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask
- Endpoint def https://stevenpcurtis.medium.com/endpoint-vs-api-ee96a91e88ca
- Venn diagram description: https://en.wikipedia.org/wiki/Venn_diagram
- What is Venn diagram: https://www.lucidchart.com/pages/tutorial/venn-diagram
- Venn diagram Python: https://towardsdatascience.com/how-to-create-and-customize-venn-diagrams-in-python-263555527305
- PYPI : https://pypi.org/classifiers/
- Git config settings: https://stackoverflow.com/questions/68775869/support-for-password-authentication-was-removed-please-use-a-personal-access-to
- Build python packages: https://www.freecodecamp.org/news/build-your-first-python-package/
-