Connecting to GitHub

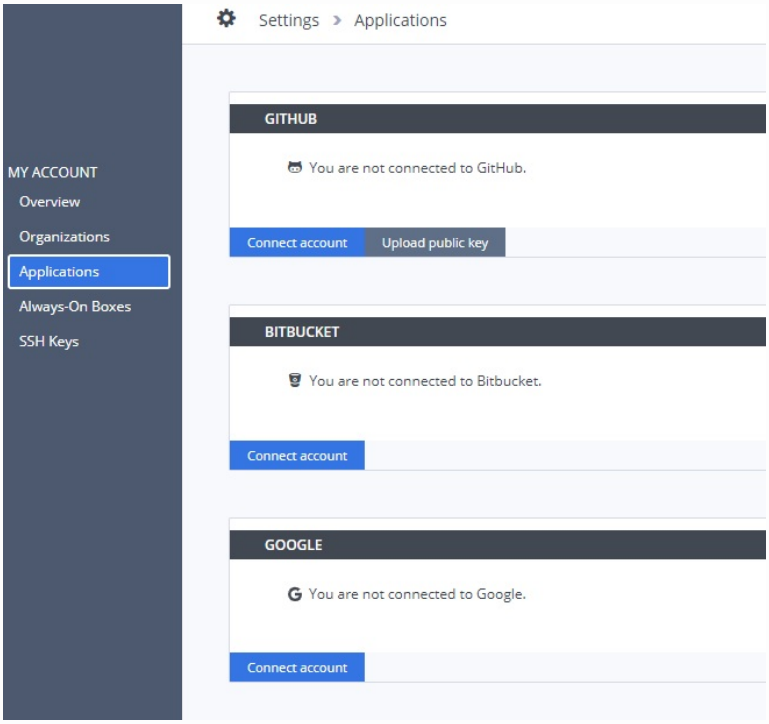
Connecting to GitHub

We are going to build a Python package that you can upload to your GitHub repository. If you do not yet have an account, please create one now. We are going to clone a repository that will contain the code for this project.

Connecting GitHub and Codio

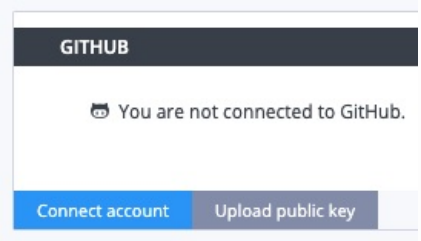
You need to connect GitHub to your Codio account. This only needs to be done one time.

- In your Codio account, click on your username
- Click on **Applications**



The image depicts all of the services you can connect to your Codio account. GitHub is the topmost option.

- Under GitHub, click on **Connect account**



The image depicts the buttons associated with connecting your GitHub account to Codio. The first button says to connect GitHub. The second button says to upload a public key.

- You will be using an SSH connection, so you need to click on **Upload public key**

Fork the Repository

- Go to the sw_characters repository. This repository is the starting point for your project.
- Click on the "Fork" button in the top-right corner.
- Click the green "Code" button.
- Copy the SSH information. It should look something like this:

```
git@github.com:<your_github_username>/sw_characters.git
```

info

Important

If you do not use the SSH information, you will have to provide your username and Personal Access Token (PAT) to GitHub each time you push or pull from the repository. See this documentation for setting up a PAT for your GitHub account.

In the Terminal

- Clone the repository. Your command should look something like this:

```
git clone git@github.com:
        <your_github_username>/sw_characters.git
```

- You should see a `sw_characters` directory appear in the file tree.

You are now ready to start the project.

Star Wars Characters Project

Working with a Star Wars API

In many examples for installing package, the `requests` package was often used. This package allows you to more easily talk to web pages. We are going to use this package to query a Star Wars API and print a short description of a character that looks like the output below. In addition, the script should be able to handle a list of characters returned from the API.

```
Luke Skywalker is from the planet Tatooine. They appear in the
following films:
  * A New Hope
  * The Empire Strikes Back
  * Return of the Jedi
  * Revenge of the Sith
```

Perhaps Star Wars isn't your thing, but it is a full-featured API that does not require authentication. This makes the API a useful tool to introduce these concepts with a little less complexity.

We are also going to use the `fire` package to create an easy command line interface for the end user. They can use the `--name` flag to identify the character to be searched.

```
python3 sw_character.py --name="boba"
```

Finally, we are going to use TinyDB a lightweight, document-based database to improve the performance by limiting the number of times the script needs to query the API.

Starting the Project

You may have noticed that `(base)` is at the beginning of the prompt. That means Conda is already installed. We will be using this package manager for this project. The first thing we need to do is create a virtual environment for this project.

```
conda create -n sw -y
```

Next, activate the new virtual environment.

```
conda activate sw
```

You should see `(sw)` replace `(base)` in the terminal. Then install the `requests` library with the `install` command.

```
conda install requests -y
```

Finally, change into the directory that stores the source code and create Python files for searching the API and creating the user interface.

```
touch sw_characters/search_api.py sw_characters/interface.py
```

To verify that your project is ready to go, use the `tree` command see the newly created files.

```
tree sw_characters/
```

You should see the following output for the `sw_search` directory:

```
sw_characters/
├── __init__.py
├── interface.py
└── search_api.py
```

Searching the API

Initial Search

Activate the virtual environment.

```
conda activate sw
```

While the Star Wars API is very broad, we are going to focus on just the characters from the films. You can access a specific character by referencing a specific URL and the site returns the relevant information in JSON. For example, the page below is for Luke Skywalker:

```
https://swapi.dev/api/people/1/
```

The problem is, you need to know the exact URL for each character if want to access their information. The API also provides a way to search it. This too is done with a specific URL. Replace the number with `?search=r2` where `r2` represents the search term. The API will still return information as JSON.

```
https://swapi.dev/api/people/?search=r2
```

Start by importing `requests`. Create the `search_sw` function with `luke` as the default value for `search_term`. For now, put `pass` as the body of the function. In addition, we are going to set up a space to test our code. Use a conditional to ask if we are running the script directly. Use `pass` as the body of the conditional for now.

```
import requests

def search(search_term='luke'):
  pass

if __name__ == '__main__':
  pass
```

Every search starts with the same base URL. Create the variable `base_url` that contains the search URL minus the search term. Then create the `search_url` variable that combines the `base_url` and the `search_term`

argument.

```
def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
```

The `resp` variable represents the response from searching the API. Use the `get()` method from the `requests` package and the `search_url` as the website to request. Then transform the response into JSON with the `json()` method. Python treats JSON like a dictionary, which means we can easily access the keys and values.

```
def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
```

If the API call worked and returned information, it will be stored in the `results` key. The associated value is a list of dictionaries. Use a conditional to determine if the value for `results` is an empty list or not. An empty list means the character is not found in the API. If the list is not empty, return the first element from `results`, which is a dictionary. If `results` is an empty list, return `None`.

```
def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results'][0]
  else:
    return None
```

Testing the Function

Let's test out the newly created function. In the testing conditional import the `pprint` package. This allows us to print the contents of a dictionary in an easy to read manner. Create `character` and set it to the return value from `search_sw`. By default, the function will search for `'luke'`. Use `pprint` to print `character`.

```
if __name__ == '__main__':
  import pprint

  character = search()
  pprint.pprint(character)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should see the following output:

```
{'birth_year': '19BBY',
 'created': '2014-12-09T13:50:51.644000Z',
 'edited': '2014-12-20T21:17:56.891000Z',
 'eye_color': 'blue',
 'films': ['https://swapi.dev/api/films/1/',
           'https://swapi.dev/api/films/2/',
           'https://swapi.dev/api/films/3/',
           'https://swapi.dev/api/films/6/'],
 'gender': 'male',
 'hair_color': 'blond',
 'height': '172',
 'homeworld': 'https://swapi.dev/api/planets/1/',
 'mass': '77',
 'name': 'Luke Skywalker',
 'skin_color': 'fair',
 'species': [],
 'starships': ['https://swapi.dev/api/starships/12/',
               'https://swapi.dev/api/starships/22/'],
 'url': 'https://swapi.dev/api/people/1/',
 'vehicles': ['https://swapi.dev/api/vehicles/14/',
              'https://swapi.dev/api/vehicles/30/']}
```

Now, test the function with a search term that does not exist, `'asdf'`.

```
if __name__ == '__main__':
  import pprint

  character = search('asdf')
  pprint.pprint(character)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should see the following output:

```
None
```

▼ **Code**

Your code should look like this:

```
import requests

def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results'][0]
  else:
    return None

if __name__ == '__main__':
  import pprint

  character = search('asdf')
  pprint.pprint(character)
```

Deactivate the virtual environment.

```
conda deactivate
```

Parsing the Name and Planet

Parsing the Name

Activate the virtual environment.

```
conda activate sw
```

We need the name from the JSON data. Using the `get()` method, store the value of `'name'` and return it. The name is stored as a string in the dictionary.

```
def parse_name(person):
  name = person.get('name')
  return name
```

Parsing the Planet

Unlike the name, the planet is not stored as a string. Instead, it is another URL for the Star Wars API. First, extract the planet URL which is stored under the `'homeworld'` key. Then make another API call with the `requests` package. Convert the response to JSON and return the value associated with the `'name'` key, which is a string of the planet's name.

```
def parse_planet(person):
  planet_url = person.get('homeworld')
  resp = requests.get(planet_url)
  planet = resp.json().get('name')
  return planet
```

Testing the Functions

In the testing section at the bottom, change the character search to be the default. Then create the `name` variable and pass it `character`. Print `name` and you should see `'Luke Skywalker'`.

```
if __name__ == '__main__':
  import pprint

  character = search()
  name = parse_name(character)

  pprint.pprint(name)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

Next, test the `parse_planet` function. Create the variable `planet` and set it to the function call `parse_planet`. Use `character` as the parameter. You should see `'Tatooine'` as the output.

```
if __name__ == '__main__':
  import pprint

  character = search()
  name = parse_name(character)
  planet= parse_planet(character)

  pprint.pprint(planet)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should notice that printing the name of the planet takes longer than printing the name. That is because printing the name requires only one API call. Printing the planet name requires two API calls — one for the general character information and another for the planet information. We will see how these API calls start to add up and affect the performance of the script.

▼ **Code**

Your code should look like this:

```
import requests

def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results'][0]
  else:
    return None

def parse_name(person):
  name = person.get('name')
  return name

def parse_planet(person):
  planet_url = person.get('homeworld')
  resp = requests.get(planet_url)
  planet = resp.json().get('name')
  return planet

if __name__ == '__main__':
  import pprint

  character = search()
  name = parse_name(character)
  planet= parse_planet(character)

  pprint.pprint(planet)
```

Deactivate the virtual environment.

```
conda deactivate
```

Parsing the Films

Parsing Film Titles

Activate the virtual environment.

```
conda activate sw
```

The next thing to do is to get a list of the films in which the character appears. Film titles are stored as a list of URLs. These need to be converted into a list of strings. Create the `parse_films` function and pass it the argument `person` which is the general dictionary of the character you got from the API. Set `film_urls` to the value associated with `'films'`, which will be a list of URLs. Using a list comprehension create a list of strings. Use the helper function `fetch_title` to convert the URL to a string.

```
def parse_films(person):
  film_urls = person.get('films')
  films =[fetch_title(film_url) for film_url in film_urls]
  return films
```

The `fetch_title` function calls the API with each film's URL and converts the results into JSON. Then get the value associated with the `'title'` key. Return this string back to the list comprehension above.

```
def fetch_title(url):
  film_json = requests.get(url).json()
  film_title = film_json.get('title')
  return film_title
```

Testing the Function

In the testing section, create the variable `film_list` and call the `parse_films` function. This function should return a list of strings.

```
if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)

  pprint.pprint(film_list)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should see the following output:

```
['A New Hope',
 'The Empire Strikes Back',
 'Return of the Jedi',
 'Revenge of the Sith']
```

Formatting the Film Titles

So the film titles are now stored as strings, but they are in a list. We want to have a single string with all of the film titles. To improve readability, we want each title on its own line. We also want to indent two spaces and preface the title with an `*`. Create the `format_titles` function that takes a list of movie titles. With a list comprehension, add a newline character (`\n`) to the end of each element in the list. Use the string method `join()` to create a string of all of the titles in the list, placing `'  * '` between each element. Add `'  * '` to the beginning of the string so that all elements have the same indentation before the title. Return a single string with all of the film titles, indentation, and newline characters.

```
def format_titles(titles):
  new_lines = [title + '\n' for title in titles]
  formatted_titles = '  * ' + '  * '.join(new_lines)
  return formatted_titles
```

Testing the Function

In the testing section, create the variable `title` and call the function `format_titles`. Printing this variable should display an indented movie title on each line.

```
if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)
  titles = format_titles(film_list)

  print(titles)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should see the following output:

```
  * A New Hope
  * The Empire Strikes Back
  * Return of the Jedi
  * Revenge of the Sith
```

Building the Description

So far, the functions we built helped to create the components needed to build the discussion for each character. The `person_description` function builds the final description. Using an f-string, name the character, where they are from, and the movies in which they appear. Return this string.

```
def person_description(name, planet, titles):
  description = f'{name} is from the planet {planet}. They
        appear in the following films:\n{titles}'
  return description
```

Testing the Function

In the testing section, create the `description` variable and call the `person_description` function. Pass `name`, `planet`, and `titles` to the function. Your script should print the full description of a character from Star Wars.

```
if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)
  titles = format_titles(film_list)
  description = person_description(name, planet, titles)

  print(description)
```

In the terminal, run the script to test your work.

```
python sw_characters/search_api.py
```

You should see the following output:

```
Luke Skywalker is from the planet Tatooine. They appear in the
following films:
  * A New Hope
  * The Empire Strikes Back
  * Return of the Jedi
  * Revenge of the Sith
```

▼ Code

Your code should look like this:

```
import requests

def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results'][0]
  else:
    return None

def parse_name(person):
  name = person.get('name')
  return name
```

```
def parse_planet(person):
  planet_url = person.get('homeworld')
  resp = requests.get(planet_url)
  planet = resp.json().get('name')
  return planet

def parse_films(person):
  film_urls = person.get('films')
  films =[fetch_title(film_url) for film_url in film_urls]
  return films

def fetch_title(url):
  film_json = requests.get(url).json()
  film_title = film_json.get('title')
  return film_title

def format_titles(titles):
  new_lines = [title + '\n' for title in titles]
  formatted_titles = '  * ' + '  * '.join(new_lines)
  return formatted_titles

def person_description(name, planet, titles):
  description = f'{name} is from the planet {planet}. They
        appear in the following films:\n{titles}'
  return description

if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)
  titles = format_titles(film_list)

  print(titles)
```

Deactivate the virtual environment.

```
conda deactivate
```

Fire Package

Setting Up the Command Line Interface

Activate the virtual environment.

```
conda activate sw
```

Then, install the Fire package. Fire makes creating CLIs much easier. The Fire package, however, is not found in Anaconda. It has to be installed from PyPI. Use `pip` to install `fire`.

```
pip install fire
```

In the top-left panel, import the newly installed `fire` package and the `search_api` module we just created.

```
import fire
import search_api
```

Declare the `search` function that takes the argument `name` with the default value `'luke'`. For now, have a simple print statement that let's us know the for which name the function is searching.

```
import fire
import search_api

def search(name='luke'):
  print(f'searching for {name}')
```

Fire requires that you instantiate a `fire` object in the `__name__ == '__main__'` conditional. Put the `search` function in parentheses, which exposes this to the end user. That means a user can only run the `search` function. All other functions we create will be hidden from them. When you run the script, Fire will automatically run the `search` function.

```
import fire
import search_api

def search(name='luke'):
  print(f'searching for {name}')

if __name__ == '__main__':
  fire.Fire(search)
```

Testing the Function

In the terminal, run the script to test your work. By default, `search` should look for the search term `'luke'`.

```
python3 sw_characters/interface.py
```

You should see the following output:

```
searching for luke
```

We can pass different names to the function with the `--name` flag. Change the search term to `boba`. In the terminal, run the script to test your work.

```
python sw_characters/interface.py --name="boba"
```

You should see the following output:

```
searching for boba
```

▼ **Code**

Your code should look like this:

```
import fire
import search_api

def search(name='luke'):
  print(f'searching for {name}')

if __name__ == '__main__':
  fire.Fire(search)
```

Deactivate the virtual environment.

```
conda deactivate
```

Invoking the Search

Handling Bad Searches

Activate the virtual environment.

```
conda activate sw
```

It is possible that the user enters a search term that is not a character in Star Wars. Remember, our initial search of the API returns a list of Star Wars characters or `None`. We need to be able to handle a search that returns `None`.

Modify the `search` function so that it invokes `search` from the `search_api` module. This value should be `None` or a list. If the value is a list, write `pass` as we are not concerned about a valid search (for now at least). If the value is `None`, print a message that the script cannot find that character.

```
def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    pass
  else:
    print(f'Cannot find the character "{name}"')
```

Testing Bad Searches

Next, enter the following command to call our script with a search term that is not a valid Star Wars character.

```
python3 sw_characters/interface.py --name="asdf"
```

You should see the following output:

```
Cannot find the character "asdf"
```

Handling Valid Searches

Replace `pass` with a print statement that prints the result of the `parse_char` function.

```
def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    print(parse_char(characters))
  else:
    print(f'Cannot find the character "{name}"')
```

The `parse_char` function parses the dictionary of a Star Wars character and returns the description. Basically, this function is going to call the functions we wrote in the `search_api` module.

Create variables for the character's name, home planet, list of films, the formatted string of film titles, and the final description. Use the appropriate function for each variable.

▼ **Did you notice?**
Preface all of the function calls with `search_api` so Python knows where to look. If not, Python will throw an error.

```
def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  return description
```

Testing Valid Searches

The virtual environment should already be running. If not, follow the instructions above. Run the script with the default search term.

```
python3 sw_characters/interface.py
```

You should see the following output:

```
Luke Skywalker is from the planet Tatooine. They appear in the
following films:
  * A New Hope
  * The Empire Strikes Back
  * Return of the Jedi
  * Revenge of the Sith
```

Test the function one more time with `"boba"` as the search term.

```
python3 sw_characters/interface.py --name="boba"
```

You should see the following output:

```
Boba Fett is from the planet Kamino. They appear in the
following films:
  * The Empire Strikes Back
  * Return of the Jedi
  * Attack of the Clones
```

▼ **Code**

Your code should look like this:

```
import fire
import search_api

def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    print(parse_char(characters))
  else:
    print(f'Cannot find the character "{name}"')

def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  return description

if __name__ == '__main__':
  fire.Fire(search)
```

Deactivate the virtual environment.

```
conda deactivate
```

Multiple Search Results

Only One Search Result

Activate the virtual environment.

```
conda activate sw
```

Use the string `"b"` as the search term. This should return those characters who have the letter `b` in their name.

```
python sw_characters/interface.py --name="b"
```

You should see the following output:

```
Beru Whitesun lars is from the planet Tatooine. They appear in
the following films:
  * A New Hope
  * Attack of the Clones
  * Revenge of the Sith
```

We know that `"Boba Fett"` is character with the letter `b` in his name, yet he does not show up in the search results. In fact, only one character is returned. This would be fine if the user knew the exact person for whom they want to search. If they are looking for multiple users, then we need to make some changes to our code.

Multiple Search Results

Let's look back at our original `search` function. It only returns the first element from the list associated with the `results` key. Remove the `[0]` from the return statement. We want all the search results.

```
def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results']
  else:
    return None
```

Click the link below to open the `interface.py` file. We need to change how this module parses data returned from the API since it now returns a list.

The `parse_char` function is no longer adequate because we are no longer parsing a single character. We need to parse a list of characters. Create the function `parse_char_list` that takes a list of characters. Iterate over the list. Create variables for the character's name, planet, list of films, formatted string of film titles, and the final description. Call the appropriate function from the `search_api` module. Finally print the description.

```
def parse_char_list(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    planet= search_api.parse_planet(char)
    film_list = search_api.parse_films(char)
    titles = search_api.format_titles(film_list)
    description = search_api.person_description(char_name,
        planet, titles)
    print(description)
```

Go back to the top of the file and find the `search` function. We need to replace the call to the `parse_char` function with a call to `parse_char_list` instead. We no longer need a print statement here as one is found in `parse_char_list`.

```
def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    parse_char_list(characters)
  else:
    print(f'Cannot find the character "{name}"')
```

Testing the Functions

Let's first make sure our script still works with a single character to print. This should produce the information for Luke Skywalker that we have seen already.

```
python sw_characters/interface.py
```

Now let's test our script with a list of Star Wars characters. This is going to take some time as there are several API calls per character.

```
python sw_characters/interface.py --name="b"
```

The script should return information about Beru Whitesun lars, Biggs Darklighter, Obi-Wan Kenobi, Chewbacca, Jabba Desilijic Tiure, Boba Fett, Bossk, Lobot, Ackbar, and Nien Nunb. It may work, but the slow speed means it is a terrible user experience.

▼ **Code**

Your code should look like this:

- **`search_api.py`**

```
import requests

def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results']
  else:
    return None

def parse_name(person):
  name = person.get('name')
  return name

def parse_planet(person):
  planet_url = person.get('homeworld')
  resp = requests.get(planet_url)
  planet = resp.json().get('name')
  return planet

def parse_films(person):
  film_urls = person.get('films')
```

```
  films =[fetch_title(film_url) for film_url in film_urls]
  return films

def fetch_title(url):
  film_json = requests.get(url).json()
  film_title = film_json.get('title')
  return film_title

def format_titles(titles):
  new_lines = [title + '\n' for title in titles]
  formatted_titles = '  * ' + '  * '.join(new_lines)
  return formatted_titles

def person_description(name, planet, titles):
  description = f'{name} is from the planet {planet}. They
        appear in the following films:\n{titles}'
  return description

if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)
  titles = format_titles(film_list)

  print(titles)
```

- **interface.py**

```
import fire
import search_api

def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    parse_char_list(characters)
  else:
    print(f'Cannot find the character "{name}"')

def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  return description

def parse_char_list(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    planet= search_api.parse_planet(char)
    film_list = search_api.parse_films(char)
    titles = search_api.format_titles(film_list)
    description = search_api.person_description(char_name,
        planet, titles)
    print(description)

if __name__ == '__main__':
  fire.Fire(search)
```

Deactivate the virtual environment.

```
conda deactivate
```

Improving Performance

Too Many API Calls

Activate the virtual environment.

```
conda activate sw
```

Accessing the network is a relatively slow process for a computer. At a minimum, every character will make three API calls. If they appear in more films, then the script makes more API calls.

If we want to reduce the number of API calls, then we need to store the information locally. This is the benefit of using a database. Before accessing the network, first query the database for a matching record. If it exists, read from the database. If not, get the information from the API and then add it to the database. The more searches are made, the more information is stored locally, the faster the script runs.

We are going to use TinyDB for this project. There are many databases that work well with Python. Many of the databases are more robust and faster. However, we are going to keep things simple. With TinyDB, all data is stored in a local JSON file. The database package provides you with a nice interface for accessing and modifying the information in the database.

TinyDB is found the in the `conda-forge` channel, so the installation will look a bit different.

```
conda install --channel=conda-forge tinydb
```

Implementing the Database

Once the database installed, we need to import `TinyDB` and `Query` from the `tinydb` package.

```
import fire
from tinydb import TinyDB, Query
import search_api
```

Next, instantiate our database with the `db.json` file. Python will create this file for us. Then we need to instantiate a `Query` object. This will allow us to search the database.

```
db = TinyDB('db.json')
User = Query()
```

Find the `search` function. If the API returns information, we want to check if it exists in the database. Replace the call to the `parse_char_list` function with a call to the `check_db` function.

```
def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    check_db(characters)
  else:
    print(f'Cannot find the character "{name}"')
```

The `check_db` function takes a list of Star Wars characters. Iterate over the list, extracting the name of the character. Use the `search` method on the database object to see if it storing the user name. The `search` method returns a list of database records. Determine if the list is empty. If yes, create the `description` variable by calling the `parse_char` function. This is the slow part of our script as all of the API calls will be made.

```
def check_db(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    results = db.search(User.name == char_name)
    if not results:
      description = parse_char(char)
```

If the database search does not return an empty list, extract the values associated with the `name`, `planet`, and `titles` keys. Create the `description` variable by calling `person_description` function from the `search_api` module. In short, collect the information from the database, not from the API. Finally, print the description. The print statement should not be inside the `else` branch.

```
def check_db(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    results = db.search(User.name == char_name)
    if not results:
      description = parse_char(char)
    else:
      name = results[0]['name']
      planet = results[0]['planet']
      titles = results[0]['titles']
      description = search_api.person_description(name, planet,
        titles)
    print(description)
```

The `parse_char` function is only called if the character is not in the database. Before returning the description, add a line that inserts the character name, planet, and formatted titles to the database so we no longer have to make an API call for that character.

```
def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  db.insert({'name':char_name, 'planet':planet,
        'titles':titles})
  return description
```

▼ Code

Your code should look like this:

```
import fire
from tinydb import TinyDB, Query
import search_api

db = TinyDB('db.json')
User = Query()

def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    check_db(characters)
  else:
    print(f'Cannot find the character "{name}"')
```

```
def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  db.insert({'name':char_name, 'planet':planet,
        'titles':titles})
  return description

def parse_char_list(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    planet= search_api.parse_planet(char)
    film_list = search_api.parse_films(char)
    titles = search_api.format_titles(film_list)
    description = search_api.person_description(char_name,
        planet, titles)
    print(description)

def check_db(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    results = db.search(User.name == char_name)
    if not results:
      description = parse_char(char)
    else:
      name = results[0]['name']
      planet = results[0]['planet']
      titles = results[0]['titles']
      description = search_api.person_description(name, planet,
        titles)
    print(description)

if __name__ == '__main__':
  fire.Fire(search)
```

Deactivate the virtual environment.

```
conda deactivate
```

Testing Performance

Increased Performance (but not at first)

This time we are going to change into the `sw_characters` directory and then activate the virtual environment. This means that when Python creates our `db.json` file, it will in the same directory as the modules.

```
cd sw_characters/ && conda activate sw
```

Use the string `"b"` as the search term. **Remember**, the database is going to be empty at first. So Python will have to do all of the API calls, which takes a long time.

```
python interface.py --name="b"
```

Now run the same exact search again:

```
python interface.py --name="b"
```

This time, it should be much faster. The results are not instantaneous, as the script still has to make the initial API call that generates the list of characters. After that, all of the information is pulled from the database. Every search for a new character search will be "slow" and saved to the database. Searching for these characters again will be much faster. Because the results are written to a file, they are persistent.

State of the Database

Let's take a look at how Python stores information in the database. There are two different views of this data. The first is the actual contents of the JSON file. Click the link below to open the database.

Notice how each record has a document ID. For example, Beru Whitesun has the ID `1`, while Biggs Darklighter has the ID `2`. **Important**, be careful about editing this file as it affects the database and the output of our script.

However, if you access the database through Python itself, you are presented with the same information in a slightly different format. Enter the command below to start the interactive Python shell. The prompt

transforms to >>> when you are in the Python shell. We can enter Python code one line at a time and it will be executed when we press ENTER. Click on the tab with the terminal and start the Python shell:

```
python
```

First, import the `interface` module. Give it the alias `i` to reduce the amount of typing we need to do.

```
import interface as i
```

Now, take a look of the contents of the database object. Use the `all()` method to return the entire database. Remember, the database is located in the `interface` module which we refer to as `i`. You do not need to use a print statement as Python will send the return value to the interpreter automatically.

```
i.db.all()
```

Instead of seeing a large dictionary, Python returns a list of dictionaries. You are not presented with the document ID for each record. You can access and manipulate data like any other list in Python.

```
characters = i.db.all()
boba = characters[5]
boba['name'].upper()
```

Exit the Python shell with the following command:

```
exit()
```

Wrapping Up

To close out this project, we need to clean up our code a bit and prepare it for others to use.

- We no longer need the `parse_char_list` function. It is not hurting anything since we no longer call it, but removing it makes our code a bit easier to read and understand. This one is really up to you. Click the link below to open the `interface.py` file so you can remove the `parse_char_list` function (if you want).

- As we prepare the project for use by others, we should remove the `db.json` file. Python will create this file if it is not present, so this won't break our script. In addition, this will reduce the size of our package when others download it. The user can populate the database with their own searches. Run the following command in the terminal:

```
rm db.json
```

- Finally we need to use Conda to create a requirements file in the `yaml` format so that it includes *all* packages installed for this project. Should you share this project with others, they can use the `yaml` file to build a virtual environment with all of the required packages with a single command.

```
conda env export > requirements.yaml
```

Verify that Conda created the `yaml` file by using the `cat` command. This sends the contents of `requirements.yaml` to the terminal.

```
cat requirements.yaml
```

Verify that you see both the default and `conda-forge` channels listed in the document:

```
channels:
  - conda-forge
  - defaults
```

Also look for the packages installed with `pip`. That would be `fire` and any of its dependencies:

```
  - pip:
    - fire==0.4.0
    - six==1.16.0
    - termcolor==1.1.0
```

▼ **Code**

Your code should look like this:

- **`search_api.py`**

```
import requests
```

```
def search(search_term='luke'):
  base_url = 'https://swapi.dev/api/people/?search='
  search_url = f'{base_url}{search_term}'
  resp = requests.get(search_url)
  resp_json = resp.json()
  if resp_json.get('results'):
    return resp.json()['results']
  else:
    return None

def parse_name(person):
  name = person.get('name')
  return name

def parse_planet(person):
  planet_url = person.get('homeworld')
  resp = requests.get(planet_url)
  planet = resp.json().get('name')
  return planet

def parse_films(person):
  film_urls = person.get('films')
  films =[fetch_title(film_url) for film_url in film_urls]
  return films

def fetch_title(url):
  film_json = requests.get(url).json()
  film_title = film_json.get('title')
  return film_title

def format_titles(titles):
  new_lines = [title + '\n' for title in titles]
  formatted_titles = '  * ' + '  * '.join(new_lines)
  return formatted_titles

def person_description(name, planet, titles):
  description = f'{name} is from the planet {planet}. They
        appear in the following films:\n{titles}'
  return description

if __name__ == '__main__':
  import pprint

  person = search()
  name = parse_name(person)
  planet= parse_planet(person)
  film_list = parse_films(person)
  titles = format_titles(film_list)
```

```
  print(titles)
```

- **interface.py**

```
import fire
from tinydb import TinyDB, Query
import search_api

db = TinyDB('db.json')
User = Query()

def search(name='luke'):
  characters = search_api.search(name)
  if characters is not None:
    check_db(characters)
  else:
    print(f'Cannot find the character "{name}"')

def parse_char(char):
  char_name = search_api.parse_name(char)
  planet= search_api.parse_planet(char)
  film_list = search_api.parse_films(char)
  titles = search_api.format_titles(film_list)
  description = search_api.person_description(char_name, planet,
        titles)
  db.insert({'name':char_name, 'planet':planet,
        'titles':titles})
  return description

def check_db(chars):
  for char in chars:
    char_name = search_api.parse_name(char)
    results = db.search(User.name == char_name)
    if not results:
      description = parse_char(char)
    else:
      name = results[0]['name']
      planet = results[0]['planet']
      titles = results[0]['titles']
      description = search_api.person_description(name, planet,
        titles)
    print(description)

if __name__ == '__main__':
  fire.Fire(search)
```

Deactivate the virtual environment.

```
conda deactivate
```

Pushing to GitHub

Pushing to GitHub

Before leaving, add this project to your GitHub portfolio. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish Star Wars character project"
```

- Push to GitHub:

```
git push
```

Lab Challenge

Lab Challenge

Problem

Create and activate the `lab-challenge` virtual environment with 3.10.4 as the Python version.

Expected Output

Once your virtual environment is set up, run the following commands in the terminal:

```
conda activate lab-challenge
conda list
```

You should see `lab-challenge` in parentheses at the beginning of the prompt, which means the environment is active.

```
(lab-challenge) codio@janetoption-torchhydro:~/workspace$
```

Among the list of installed packages, you should see Python version 3.10.4.

```
python                    3.10.4               h12debd9_0
```