

SI 506: Last Assignment

1.0 Dates

- Available: Thursday, 14 April 2022, 4:00 PM Eastern
- Due: on or before Tuesday, 26 April 2022, 11:59 PM Eastern

! No late submissions will be accepted for scoring.

2.0 Overview

The last assignment is open network, open readings, and open notes. You may refer to code in previous lecture exercises, lab exercises, and problem sets for inspiration.

We recommend that at a minimum you bookmark the following [w3schools](#) Python pages and/or have them open in a set of browser tabs as you work on the last assignment:

- [Python keywords](#)
- [Python operators](#)
- [Python built-in functions](#)
- [Python dict methods](#)
- [Python list methods](#)
- [Python str methods](#)
- [SWAPI documentation](#)

3.0 Points

The last assignment is worth **1700** points and you accumulate points by passing a series of autograder tests.

4.0 Solo effort

! You are prohibited from soliciting assistance or accepting assistance from any person while completing the programming assignment. The code that you submit *must* be your own work. Likewise, you are prohibited from assisting any other student required to complete this assignment. This includes those attempting the assignment during the regular exam period, as well as those who may attempt the assignment at another time and/or place due to scheduling conflicts or other issues.

! If you have formed or participate in an SI 506 study group please suspend all study group activities for the duration of the last assignment.

5.0 Data

The Star Wars saga has spawned films, animated series, books, music, artwork, toys, games, fandom websites, cosplayers, scientific names for new organisms (e.g., [Trigonopterus yoda](#)), and even a Darth Vader *grotesque* attached to the [northwest tower](#) of the Washington National Cathedral. Leading US news organizations such as the [New York Times](#) cover the Star Wars phenomenon on a regular basis.

The last assignment adds yet another Star Wars-inspired artifact to the list. The data used in this assignment is sourced from the [Star Wars API](#) (SWAPI), [Wookieepedia](#), [Wikipedia](#), and the [New York Times](#).

Besides retrieving data from SWAPI you will also access information locally from the following data files:

- `clone_wars.csv`
- `clone_wars_episodes.csv`
- `nyt_star_wars_articles.json`
- `wookieepedia_droids.json`
- `wookieepedia_people.json`
- `wookieepedia_planets.csv`
- `wookieepedia_starships.csv`

6.0 Files

In line with the weekly lab exercises and problem sets you will be provided with a number of files:

1. `swapi.md`: assignment instructions
2. `swapi.py`: script including a `main()` function and other definitions and statements
3. `sw_utils.py`: module imported by `swapi.py` that contains utility functions and constants
4. One or more `*.csv` and/or `*.json` files that contain assignment data
5. One or more `fxt_*.json` test fixture files that you must match with the files you produce

Please download the assignment files from Canvas Files as soon as they are released. This is a timed event and delays in acquiring the assignment files will shorten the time available to engage with the challenges. The clock is not your friend.

! *DO NOT* modify or remove the scaffolded code that we provide in the Python script or module files unless instructed to do so.

6.1 `swapi.py`

The `swapi.py` "script" file contains function definitions that you will implement along with a `main()` function that serves as the entry point for the script (also known as a program).

From `main()` you will implement the script's workflow that will involve the following operations:

- Import modules
- Access data from local files (`*.csv`, `*.json`)
- Retrieve items from a local cache and/or issue HTTP GET requests to the remote SWAPI service
- Call functions and perform computations
- Assign values to specified variables
- Write data to local files encoded as JSON

Implementing functions involves replacing the placeholder `pass` statement with working code. You may be asked to add missing parameters to function and/or method definitions. You may also be asked to define a function in its entirety (less the Docstring) as directed per the instructions.

6.2 `sw_utils.py`

The `swapi_utils.py` module contains both constants (e.g., `SWAPI_ENDPOINT = 'https://swapi.py4e.com/api'`) and functions, a number of which you will be asked to implement. You will import the utilities module into `swapi.py` so that you can access its statements and definitions in your program.

6.3 CSV and JSON files

We will supply you with CSV and JSON files that include data for use in your program. Locate the files in the *same directory* where you place your templated script and module files.

6.4 Test fixture JSON files

You will also be supplied with a set of test fixture JSON files (prefixed with `fxt_`). Each represents the correct file output that *must* be generated by the program you write. Use them at periodic intervals to compare your current output against the expected output.

! Your output **must** match the fixture files line-for-line and character-for-character. Review these files; they constitute the answer keys and should be utilized for comparison purposes as you work your way through the assignment.

7.0 Challenges

The last assignment comprises a maximum of twenty (20) challenges. The teaching team recommends that you complete each challenge in the order specified in the instructions.

Certain challenges can be solved with a single line of code. Others may involve writing several lines of code including implementing one or more functions in order to solve the challenge.

The challenges cover all topics introduced between weeks 01 and 14 and outlined in the [syllabus](#) and described in more detail below:

- Basic syntax and semantics
 - Values (objects), variables, variable assignment
 - Expressions and statements
 - Data types
 - numeric values (`int`, `float`)
 - boolean values (`True`, `False`)
 - `NoneType` object (`None`)
 - sequences: `list`, `range`, `str`, `tuple`
 - associative array: `dict`
 - Operators: arithmetic, assignment, comparison, logical, membership
 - Escape sequences (e.g., newline `\n`)
- String formatting: formatted string literal (f-string)
- Data structures
 - Sequences
 - Indexing and slicing (subscript notation)
 - List creation, mutation (add, modify, remove elements), and element unpacking
 - `list` methods
 - Tuple packing (creation) and unpacking

- `tuple` methods
- `str` methods
- String, list, and tuple concatenation
- Associative array
 - Dictionary creation (add, modify, remove key-value pairs), subscript notation
 - `dict` methods
- Working with nested lists, tuples, and dictionaries
- `del` statement
- Control flow
 - Iteration
 - `for` loop, `for i in range()` loop, `while` loop
 - Accumulator pattern
 - Counter usage (e.g. `count += 1`)
 - `break` and `continue` statements
 - Nested loops
 - Conditional execution
 - `if`, `if-else`, `if-elif-else` statements
 - Truth value testing (`if < object >:`)
 - Compound conditional statements using the logical operators `and` and `or`
 - Negation with `not` operator
- Functions
 - Built-in functions featured in lecture notes/code, lab exercises and problem sets
 - Literacy/competency
 - `print()`
 - `len()`
 - `type()`, `isinstance()`
 - `int()`, `float()`
 - `min()`, `max()`, `sum()`
 - `round()`
 - `open()`
 - Awareness
 - `dict()`, `dir()`, `id()`, `enumerate()`, `list()`, `slice()`, `str()`, `tuple()`
 - User-defined functions
 - Defining a function with/without parameters, parameters with default values, and with/without a return statement
 - Calling a function and passing to it arguments by position and/or keyword arguments
 - Calling a function or functions from within another function's code block
 - Assigning a function's return value to a variable
 - Reading and interpreting function Docstrings
 - `main()` function (entry point for program or script)
 - Variable scope (global and local variables)
- Files read / write
 - `with` statement
 - Reading from and writing to *.txt files
 - `file_obj.read()`, `file_obj.readline()`, `file_obj.readlines()`
 - Reading from and writing to *.csv files

- `csv` module import; `csv.reader()`, `csv.writer()`, `DictReader`, `DictWriter`
- Reading from and writing to *.json files
 - `json` module import; `json.load()`, `json.dump()`
- Creating absolute and relative filepaths with `os.path`
- Modules
 - Importing modules
 - Creating and using custom modules
- Web API
 - `requests` module installation (via pip) and use
 - HTTP GET request/response
 - JSON payloads
 - caching
- Exceptions, exception handling, and debugging
 - `try/except` statements

8.0 A note on code styling

The auto grader will include tests that check whether or not your code adheres to Python styling guidelines relative to the [use of whitespace](#) in certain expressions and statements. The goal is to encourage you to write code that adheres to the Python community's styling practices. Doing so enhances code readability and aligns you with other Python programmers.

In particular, always surround the following operators on either side with a single space:

- assignment (`=`)
- augmented assignment (`+=`, `-=`, etc.)
- comparisons (`==`, `<`, `>`, `!=`, `<=`, `>=`, `in`, `not in`, `is`, `is not`)
- Booleans (`and`, `or`, `not`).

```
# Correct
var = search_entities(entities, search_term)

# Incorrect
var=search_entities(entities, search_term)

# Correct
count += 1

# Incorrect
count+=1
```

Note however that an exception exists with respect to function parameters and arguments. Do *not* surround the assignment operator with spaces when either:

1. defining a parameter with a default value
2. passing a keyword argument

```
# Correct
def create_email_address(uniqname, domain='umich.edu'):
    """TODO"""
    return f"{uniqname}@{domain}"

# Incorrect
def create_email_address(uniqname, domain = 'umich.edu'):
    """TODO"""
    return f"{uniqname}@{domain}"


# Correct
email_address = create_email_address(uniqname='anthwhyte',
domain='gmail.com')

# Incorrect
email_address = create_email_address(uniqname = 'anthwhyte', domain =
'gmail.com')
```

9.0 Debugging

As you write your code take advantage of the built-in `print` function, VS code's debugger, and VS Codes file comparison feature to check your work and debug your code.

9.1 The built-in `print` function is your friend

 as you work through the challenges make frequent use of the built-in `print()` function to check your variable assignments. Recall that you can call `print` from inside a function, loop, or conditional statement. Use f-strings and the newline escape character `\n` to better identify the output that `print` sends to the terminal as expressed in the following example.


```
print(f"\nSOME_VAL = {some_val}")
```

9.2 VS Code debugger

You can also use the debugger to check your code. If you have yet to configure your debugger see the instructions at <https://si506.org/resources/>. You can then set breakpoints and review your code in action by "stepping over" lines and "stepping into" function calls.

9.3 Compare files

Each test fixture CSV and JSON file (prefixed with `fixt-`) represents the correct file output that *must* be generated by the program you write. Use them at periodic intervals to compare your current output against the expected output.

 In VS Code you can compare or "diff" the file you generate against the appropriate test fixture file. After calling the `write_csv` function and generating a new file do the following:

1. Hover over the file with your cursor then right click and choose the "Select for Compare" option.
2. Next, hover over the appropriate test fixture file then right click and choose the "Compare with Selected" option.
3. Lines highlighted in red indicate character and/or indentation mismatches. If any mismatches are encountered close the comparison pane, revise your code, regenerate your file, and compare it again to the test fixture file. Repeat as necessary until the files match.

! Your output **must** match each fixture file line-for-line and character-for-character. Review these files; they are akin to answer keys and should be utilized for comparison purposes as you work your way through the assignment.

10.0 Gradescope submissions

You may submit your problem solution to Gradescope as many times as needed before the expiration of the exam time. Your **final** submission will constitute your exam submission.

! You *must* submit your solution file to *Gradescope* before the expiration of exam time. Solution files submitted to the teaching team after the expiration of exam time will receive a score of zero (0).

11.0 auto grader / manual scoring

The autograder runs a number of tests against the Python file you submit, which the autograder imports as a module so that it can gain access to and inspect the functions and other objects defined in your code. The functional tests are of two types:

1. The first type will call a function passing in known argument values and then perform an equality comparison between the return value and the expected return value. If the function's return value does not equal the expected return value the test will fail.
2. The second type of test involves checking variable assignments in `main()` or expressions in other functions. This type of test evaluates the code you write, character for character, against an expected line of code using a [regular expression](#) to account for permitted variations in the statements that you write. The test searches `main()` for the expected line of code. If the code is not located the test will fail.

If you are unable to earn full points on the assignment the teaching team will grade your submission **manually**. Partial credit **may** be awarded for submissions that fail one or more autograder tests if the teaching team (at their sole discretion) deem a score adjustment warranted.

If you submit a partial solution, feel free to include comments (if you have time) that explain what you were attempting to accomplish in the area(s) of the program that are not working properly. We will review your comments when determining partial credit.