SI 506: Lecture 20

Topics

1. Challenges

Vocabulary

- **Class**: "A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class." Python Official Documentation.
- **Composition**: Pattern that involves combining object types in order to create a *composite* type that models a "has a" relationship between the composite and one or more *component* objects (e.g., `Automobile` has an `Engine`; `Bicycle` has a `Crankset`, `Handlebar`, `Wheelset`, `Pedal` (2x), `Seat`, etc.).
- **Instance**: An individual object whose type is defined by the class by which it was instantiated or created.
- **Instance variable**: An variable and value bound to a specific instance of a class.
- **Instance method**: A function defined by a class and bound to a specific instance of a class.
- **self**: A variable that represents an instance of a class.

Previous

- **API**: Application Programming Interface that species a set of permitted interactions between systems.
- **Argument**. A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Boolean**. A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Built-in Function**. A function defined by the Standard Library that is always available for use.
- **Caller**. The initiator of a function call.
- **Conditional Statement**. A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Deep copying**. For a given mutable object (e.g., `list`) constructs a new compound object and recursively *copies* into it objects found in the original.
- **Dictionary**. An associative array or a map, wherein each specified value is associated with or mapped to a defined key that is used to access the value.
- **Expression**. An accumulation of values, operators, and/or function calls that return a value. `len(< some_list >)` is considered an expression.
- **f-string**. Formatted string literal prefixed with `f` or `F`.
- **File Object**. An object that provides a file-oriented application programming interface (API) to a either a text file, binary file (e.g., image file), or a buffered binary file. File objects include read and write methods for interacting with a file stored locally or remotely.
- **Flow of execution**. The order in which statements in a program are executed. Also referred to as *control flow*.
- **Function**. A defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept

arguments from the caller in order to perform a computation. A function can also be designed to return a computed value. Functions are considered "first-class" objects in the Python eco-system.

- **HTTP**: The Hypertext Transport Protocol is an application layer protocol designed to facilitate the distributed transmission of hypermedia. Web data communications largely depends on HTTP.
- **Immutable**. Object state cannot be modified following creation. Strings are immutable.
- **Iterable**. An object capable of returning its members one at a time. Both strings and lists are examples of an iterable.
- **Iteration**. Repetition of a computational procedure in order to generate a possible sequence of outcomes. Iterating over a `list` using a `for` loop is an example of iteration.
- **JSON**: Javascript Object Notation, a lightweight data interchange format.
- **Method**. A function defined by and bound to an object. For example the `str` type is provisioned with a number of methods including `str.strip()`.
- **Mutable**. Object state can be modified following creation. Lists are mutable.
- **Nested Loop**. A `for` or `while` loop located within the code block of another loop.
- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Parameter**. A named entity in a function or method definition that specifies an argument that the function or method accepts.
- **Querystring**: That part of a Uniform Resouce Locator (URL) that assigns values to specified parameters.
- **Resource**: A named object (e.g., document, image, service, collection of objects) that is both addressable and accessible via an API.
- **Scope**. The part of a script or program in which a variable and the object to which it is assigned is visible and accessible.
- **Sequence**. An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed.
- **Shallow copying**. For a given mutable object (e.g., `list`) constructs a new compound object but inserts *references* (rather than copies) into it of objects found in the original. The `list.copy()` returns a shallow copy of the original list.
- **Slice**. A subset of a sequence. A slice is created using the subscript notation `[]` with colons separating numbers when several are given, such as in `variable_name[1:3:5]`. The bracket notation uses slice objects internally.
- **Statement**. An instruction that the Python Interpreter can execute. For example, assigning a variable to a value such as `name = 'arwhyte'` is considered a statement.
- **Truth Value**. In Python any object can be tested for its truth value using an `if` or `while` condition or when it is used as an operand in a Boolean operation.
- **Tuple**. An ordered sequence that cannot be modified once it is created.
- **Tuple packing**. Assigning items to a tuple.
- **Tuple unpacking**. Assigning tuple items to an equal number of variables in a single assignment. A `list` can also be unpacked.
- **URI**: Uniform Resource Identifier that identifies unambiguously a particular resource.
- **URL**: Uniform Resource Locator is a type of URI that specifies the *location* of a resource on a network and provides the means to retrieve it.
- **URN**: Uniform Resource Name is a type of URI that provides a unique identifier for a resource but does not specify its location on a network.

1.0 Challenges

1.1 Challenge 01

Task: Implement the `Film` class.

1. Call `read_json` and retrieve the list of Star Wars films from `swapi_films.json`. Assign the return value to a variable named `films_data`.

2. Implement the `Film` class. The `Film` class includes a **class variable** named `franchise` with an assigned value of "Star Wars". You access class variables using dot notation (`.`) but unlike instance variables that are prefixed by `self` class variables are prefixed by the class name:

   ```
   Film.franchise
   ```

3. Implement the "dunder" `__init__` method specifing the following parameters that *must* be passed by the caller to initialize (e.g., create) a `Film` instance:

   - title
   - episode_id
   - release_date

4. Add a fourth *optional* instance variable named

   - audience_rating

   This additional instance variable can only be set *after* a `Film` instance is instantiated (in other words *do not* include it in the function's parameter list). Assign it a value of `None`.

5. Implement the "dunder" `__str__` method. Return the following formatted string to the caller:

   ```
   < franchise >: < title > (Episode < episode_id >)
   ```

6. Implement a `jsonable` method. Return a dictionary that includes the following key-value pairs:

   ```
   {
       'title': < val >,
       'episode_id': < val >,
       'release_date': < val >,
       'audience_rating': < val >
   }
   ```

1.2 Challenge 02

Task: Read `swapi_films.json`. Convert dictionaries to `Film` instances and add each instance to an accumulator dictionary.

1. Call `read_json` and retrieve the list of films in `swapi_films.json`. Assign the return value to a variable named `films_data`.

2. Create an "accumulator" dictionary named `films`. Loop over `films_data` and for each film dictionary use it's data to create a `Film` instance. Then assign each `Film` instance to the "accumulator" dictionary utilizing the film instance's title as the key and the film instance as the value.

   ```
   {
         { '< title >': < Film >}
         . . .
   }
   ```

1.3 Challenge 03

Task: Read `rotten_tomatoes-star_wars.json`. Convert dictionaries to `Film` instances and add each instance to an accumulator dictionary.

❗ Given time constraints, the `AudienceRating` class is implemented fully.

1. Call `read_json` and retrieve the list of ratings in `rotten_tomatoes-star_wars.json`. Assign the return value to a variable named `ratings_data`.

2. Create an "accumulator" dictionary named `audience_ratings`. Loop over `ratings_data` and for each ratings dictionary use it's data to create a `AudienceRating` instance. Then assign each `AudienceRating` instance to the "accumulator" dictionary utilizing the audience rating instance's title as the key and the `AudienceRating` instance as the value.

   ```
   {
         { '< title >': < AudienceRating >}
         . . .
   }
   ```

1.4 Challenge 04

Task: Loop over the `films` keys and assign to each `Film` instance the appropriate `AudienceRating` instance in the `audience_ratings` dictionary.

1. Loop over the `film` keys. Inside this loop implement another loop that loops over the `audience_ratings` values. Utilize the outer loop's key value to assign each `AudienceRating` instance to the appropriate `Film.audience_rating` instance variable.

   💡 match on the title between the two dictionaries.

2. Create an accumulator listed named `writeable = []`. Loop over the `films` dictionary and append a JSON-friendly dictionary representation of each `Film` instance to `writeable`.

3. Call `write_json` and write `writeable` to a JSON file named `films_ratings.json`.

1.5 Challenge 05

Task: Add a method named `get_audience_positive_rating` to the `Film` class. Refactor `Film.jsonable()` to ensure that a JSON-friendly dictionary representation of `AudienceRating` if an instance has been assigned to `Film.audience_rating` instance variable. Return a list of `Film` instances from `films` sorted by each film's positive audience rating. Serialize the list as JSON and write it to a file.

1. Implement a new `Film` method named `get_audience_positive_rating`. The method defines no parameters (other than `self`) and returns the `Film` instance's `audience_rating` `positie_rating` value.

2. *Refactor* (e.g., modify) `Film.jsonable()` so that a JSON-friendly dictionary representation of the `AudienceRating` instance assigned to `Film.audience_rating` can be returned *if* an `AudienceRating` instance has been assigned to the instance variable.

3. Convert `films` dictionary values to a `list` and assign the return value to a variable named `film_rankings`.

4. **BONUS**: Sort the `film_rankings` list method employing an anonymous `lambda` function that sorts the list by each film's positive audience rating.

   A `lambda` is

   > an anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`.

   💡 `lambda` functions assigned to the optional `key` argument can be passed to `list.sort()` or the built-in function `sorted()` in order to override the default sort order.

   ```
   film_rankings.sort(key=lambda film:
   film.audience_rating.positive_rating, reverse=True)

   # Alternative (built-in sorted() function)
   film_rankings = sorted(
       film_rankings,
       key=lambda film: film.audience_rating.positive_rating,
       reverse=True
       )
   ```

5. Create an accumulator listed named `writeable = []`. Loop over the `film_rankings` list and append a JSON-friendly dictionary representation of each `Film` instance to `writeable`.

6. Call `write_json` and write `writeable` to a JSON file named `films_ranked.json`.