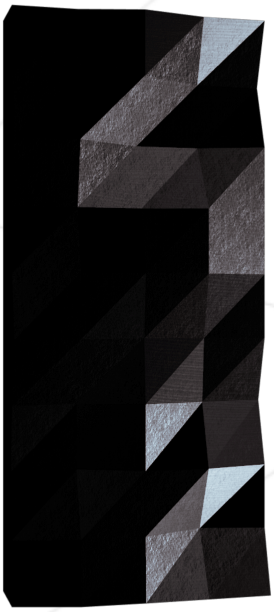


Structured and Object-Oriented Programming with Python

Module 5

Summary: In this module, you will explore class abstractions, data persistence, variable functions, and collection manipulation.

Version: 1.1

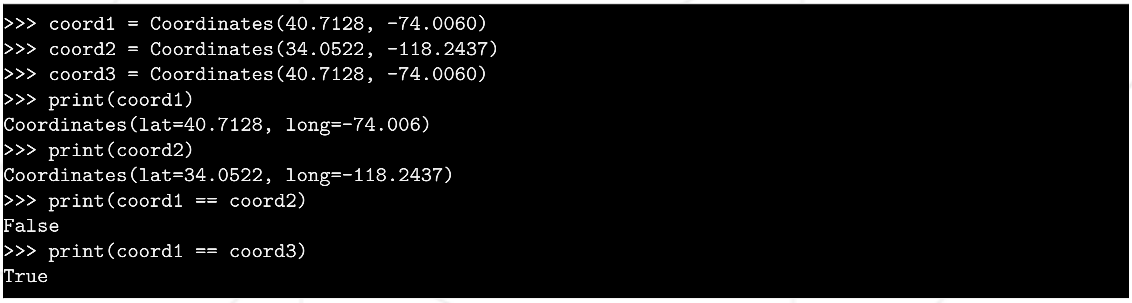
Contents

Chapter I

	Exercise 01
Class is Easy	
Turn-in directory: *ex*01/	
Files to turn in: `coordinates.py`	
Allowed functions: `standard library, chardet`	

1. Create a common class with attributes `lat` and `long` as `float`.

2. Your class should behave as follows:



```
>>> coord1 = Coordinates(40.7128, -74.0060)
>>> coord2 = Coordinates(34.0522, -118.2437)
>>> coord3 = Coordinates(40.7128, -74.0060)
>>> print(coord1)
Coordinates(lat=40.7128, long=-74.006)
>>> print(coord2)
Coordinates(lat=34.0522, long=-118.2437)
>>> print(coord1 == coord2)
False
>>> print(coord1 == coord3)
True
```

Chapter II

	Exercise 02
Dataclass is Easier	
Turn-in directory: *ex*02/	
Files to turn in: `coordinates.py`	
Allowed functions: `standard library`	

`@dataclass` [https://docs.python.org/3/library/dataclasses.html] is a decorator that automatically generates special methods in classes, such as `__init__`, `__repr__`, and `__eq__`. Instead of defining attributes within the `__init__` method using `self`, as in a simple class, in a dataclass you declare the attributes directly in the class, with type annotations.

1. Rewrite your class using a `@dataclass`.

2. The behavior should be exactly the same:

```
>>> coord1 = Coordinates(40.7128, -74.0060)
>>> coord2 = Coordinates(34.0522, -118.2437)
>>> coord3 = Coordinates(40.7128, -74.0060)
>>> print(coord1)
Coordinates(lat=40.7128, long=-74.006)
>>> print(coord2)
Coordinates(lat=34.0522, long=-118.2437)
>>> print(coord1 == coord2)
False
>>> print(coord1 == coord3)
True
```

You should be able to answer the differences between a common class and a dataclass, and which methods are implemented automatically.

Explore ‘@dataclass(options)‘

Chapter III

	Exercise 03
Data Persistence	
Turn-in directory: *ex*03/	
Files to turn in: `movies.py`	
Allowed functions: `Standard library, sqlalchemy`	

In this exercise we will use sqlalchemy. Using an ORM maps a Python class to a database table, abstracting the need for more complex implementations.

1. Create a class called `Base`, which inherits from `DeclarativeBase` from `sqlalchemy`.

    - The `Base` class should be empty, and can be implemented as below. It will be useful in the future to implement common configurations for all classes, and facilitate the relationship between them.

    ```
    class Base(DeclarativeBase):
        """future implementation"""
    ```

2. Create a class called `Movie`, which inherits from `Base`:

    - It is associated with the `movies` table.
    - It has the fields: `id`, `title`, `director`, `year`, `rating`

3. Create a program that:

    - On startup, creates a database called `movielist.sqlite` if it doesn't already exist.
    - On startup, creates the `movies` table in the database, from the `Movie` class, if it doesn't already exist.
    - If called with the argument `load <csv>`, loads the data from a `.csv` file into the database.
    - If called with the argument `show <name>`, displays the first movie whose title contains the specified name (case-insensitive).

4. Expected behavior:

```
?> python main.py load catalogue.csv
Importing movie catalog...
42 movies loaded into the database

?> python main.py show "The Matrix"
The Matrix (Wachowski, 1999) - Rating: 8.7
```

Search for ‘csv.DictReader‘ for reading csv files.

Duplicate IDs should be ignored.

Chapter IV

	Exercise 04
Functions with Variable Arguments	
Turn-in directory: *ex*04/	
Files to turn in: `arg_inspector.py`	
Allowed functions: `Standard libraries`	

You must create a function that analyzes and returns information about any arguments that are passed to it.

1. Create a function called `inspect_arguments` that accepts any number of positional and named arguments. Its prototype should be:

   ```
   from typing import Any
   def inspect_arguments(*args: Any, **kwargs: Any) -> dict[int | str, Any] | None:
       # your code
   ```

2. The function should return a dictionary where the keys are the indices (for positional arguments) or names (for named arguments) and the values are the values of the arguments.

   - For example, the first positional argument will have key 0, the second will have key 1, etc.

   If no arguments are provided, the function should return `None`.

Expected output:

```
>>> from arg_inspector import inspect_arguments
>>> inspect_arguments(42, "hello", True, name="Alice", age=30, language="Python")
{0: 42, 1: 'hello', 2: True, 'name': 'Alice', 'age': 30, 'language': 'Python'}
>>> inspect_arguments()
None
>>> inspect_arguments(1, 2, 3)
{0: 1, 1: 2, 2: 3}
>>> inspect_arguments(city="Paris", country="France")
{'city': 'Paris', 'country': 'France'}
```

More information about types can be found here.

Chapter V

Validation

- If you have reached this point and completed the exercises rigorously, you can already validate this module with a final result of 80% and already possess the necessary knowledge to advance to the following modules.

- The next exercise allows you to achieve a result of 100%. Evaluate the completion of these exercises considering the ease/difficulty encountered in completing the previous exercises. Seek a balance between challenging yourself and moving on to the next modules.

Chapter VI

	Exercise 05
People Worth Knowing	
Turn-in directory: *ex*05/	
Files to turn in: `persons_of_interest.py`	
Allowed functions: `Standard libraries`	

1. Create a function that receives a dictionary with names and birth years and returns a list with the full names, a predetermined text, and the birth year.

2. The data returned by the function must be sorted by birth year.

Expected output:

```
>>> from persons_of_interest import famous_births
>>> scientists = {
...    "ada": { "name": "Ada Lovelace", "year_of_birth": "1815" },
...    "cecilia": { "name": "Cecila Payne", "year_of_birth": "1900" },
...    "lise": { "name": "Lise Meitner", "year_of_birth": "1878" },
...    "grace": { "name": "Grace Hopper", "year_of_birth": "1906" }
...}
>>> for scientist in famous_births(scientists):
... print(scientist)
Ada Lovelace is a great scientist born in 1815.
Lise Meitner is a great scientist born in 1878.
Cecila Payne is a great scientist born in 1900.
Grace Hopper is a great scientist born in 1906.
```

`‘sorted‘, f-string`

Chapter VII

Peer Review and Submission

- Submit your project to your *Git* repository available on the project page on the intranet.
- Only the work within your repository will be evaluated during the defense. Don't hesitate to double-check your file and folder names to ensure they are correct.
- At the time of evaluation, the evaluator will go to the workstation of the student being evaluated to perform the tests. A clone of the repository will be made in a new folder, and these are the files that will be evaluated.