Project Documentation

File: main.py

This is a Python file containing two classes, `Hero` and `Archer`.

The `Hero` class has three methods: `__init__`, `get_name`, and `get_health`. The `__init__` method initializes the object with a name and health. The `get_name` method returns the name of the hero, and the `get_health` method returns the current health of the hero.

The `Archer` class inherits from the `Hero` class and adds an additional attribute, `__num_arrows`. It also defines a new method, `shoot`, which takes a target object as an argument and reduces the number of arrows by 1. If there are not enough arrows to shoot, it raises an exception. Otherwise, it subtracts 10 health from the target object.

The class hierarchy is as follows:

* `Archer` (derived from) `Hero`
* `Hero` (base class)

This code file defines two classes, `Hero` and `Archer`, where `Archer` inherits from `Hero`. The `Archer` class adds a new attribute, `__num_arrows`, and defines a new method, `shoot`. The `Hero` class has three methods: `__init__`, `get_name`, and `get_health`.

File: main_test.py

This code file contains a Python script for testing the `shoot()` method of an `Archer` class. The script defines two classes, `Hero` and `Archer`, which are used in the tests.

The script also defines several test cases, which are stored in the `test_cases` list. Each test case is a tuple containing a hero object, an archer object, and an expected result after calling the `shoot()` method on the archer object with the given hero object as an argument. The `expected_err` variable contains the expected error message, if any, that should be raised when calling the `shoot()` method.

The `test()` function takes a hero and an archer object as arguments and calls the `shoot()` method on the archer with the given hero as an argument. It then checks if the result of the `shoot()` method matches the expected result, and if an error message is raised, it checks if it matches the expected error message. If all checks pass, it prints "Pass" to the console, otherwise it prints "Fail".

The script also defines a `main()` function that runs the tests defined in the `test_cases` list. It iterates over each test case and calls the `test()` function with the corresponding hero and archer objects. If all tests pass, it prints "============= PASS ==============" to the console, otherwise it prints "============= FAIL =============".

Finally, the script checks if the `__RUN__` variable is in the global scope, if so, it sets the `test_cases` list to the `run_cases` list. This allows the user to run only the test cases that are relevant for the current context.

Overall, this code file is a good example of how to write unit tests for a Python class that implements a method with side effects. The tests cover different scenarios and edge cases to ensure that the

`shoot()` method behaves correctly in various situations.