

TASK 1

The manual approach is more efficient for the given task: "sort a list of dictionaries by a specific key." It is concise and straightforward, utilizing the basic sorted() function with a lambda. It has low overhead and clearly communicates its intention.

The AI-recommended code is more robust and feature-rich. It has optional arguments for reverse sorting and managing missing keys with a default value. This makes it much more reusable and less vulnerable to runtime problems in real-world applications where data may be inconsistent.

In terms of raw performance for the specific task, the manual version is marginally faster due to its simplicity. However, the AI version's additional safety checks (such as `dict.get()`) have a negligible performance impact for the majority of usage scenarios. The AI version is ultimately more useful for production code because of its error handling and flexibility, making it the more valued solution despite being slightly more difficult.

TASK 2(auto)

AI dramatically improves test coverage over manual testing by automating the generation, execution, and maintenance of test cases. AI-powered solutions can intelligently recognize UI elements, create tests for complex user flows, and suggest edge situations that a manual tester may overlook. They can automatically modify locators as the UI changes, reducing script breakage.

Furthermore, AI can run a large number of test combinations (for example, multiple browsers, screen sizes, and data inputs) far faster and more consistently than humans. It excels at regression testing, which ensures that new code does not break current functionality within the application. This scalability and robustness enable teams to have greater confidence in software quality and detect errors earlier in the development cycle, which is typically impossible with traditional manual testing methods.

```
C:\> Users > Mwamduu > Desktop > PL> PYTHON> 3 > auto.py ->
10 class AllignedTest(unittest.TestCase):
11     def test_valid_login(self):
12         sign_in_button.click()
13
14         # Wait for and verify success message
15         success_message = self.wait.until(
16             EC.presence_of_element_located((By.CSS_SELECTOR, ".flash.success"))
17         )
18         self.assertIn("You logged into a secure area!", success_message.text)
19
20
21 def test_invalid_login(self):
22     """Test logging in with invalid credentials."""
23     driver = self.driver
24     username_field = self.wait.until(EC.presence_of_element_located((By.ID, "username")))
25     password_field = driver.find_element(By.ID, "password")
26     login_button = driver.find_element(By.XPATH, "//button[@type='submit']")
27
28     username_field.send_keys("invalid_user")
29     password_field.send_keys("wrong_password")
30     login_button.click()
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

PYTHON 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\mwamduu & C:\Users\mwamduu\AppData\Local\Programs\Python\Python311\python.exe c:/Users/mwamduu/Desktop/PL/ PYTHON/3/auto.py ..

Run 2 tests in 40.262s

OK



The screenshot shows a web browser window with the URL the-internet.herokuapp.com/login. The page title is "Login Page". It contains a form with two input fields: "Username" and "Password", both of which have placeholder text ("invalid_user" and "wrong_password" respectively). Below the form is a large blue "Login" button with a white key icon.

Chrome is being controlled by automated test software.

This is where you can log into the secure area. Enter `tomsmith` for the username and `SuperSecretPassword!` for the password. If the information is wrong you should see error messages.

Powered by Elemental Selenium

TASK 3(analytics)

