**Name:** Alex Villegas

**Date:** 10/08/2023

ACCEPTANCE TESTING

In this lab we are going to talk about acceptance testing and how we can use it in a python program. Acceptance testing is a method of software testing where a system is tested for acceptability. The major aim of this test is to evaluate the compliance of the system with the business requirements and assess whether it is acceptable for delivery or not.

Acceptance testing plays a crucial role in software development, serving several purposes that contribute to the overall quality and success of a product. One key function of acceptance testing is to uncover defects that might have been missed during earlier phases of testing, such as functional testing. By subjecting the product to real-world scenarios and user interactions, acceptance testing helps identify any issues that could impact its functionality, performance, or user experience.

Furthermore, acceptance testing evaluates how well the product aligns with the intended requirements and user needs. It ensures that the delivered product fulfills the expectations of the customers and meets their specific requirements. This iterative process of involving users in testing provides valuable feedback that aids in refining the product, enhancing its performance, and optimizing the user experience.

The benefits of acceptance testing are manifold. Firstly, it facilitates direct collaboration between the project team and the users. This interaction not only allows for the identification of further user requirements but also nurtures a sense of ownership among users. The ability to automate test execution streamlines the testing process, increasing efficiency and reducing human errors.

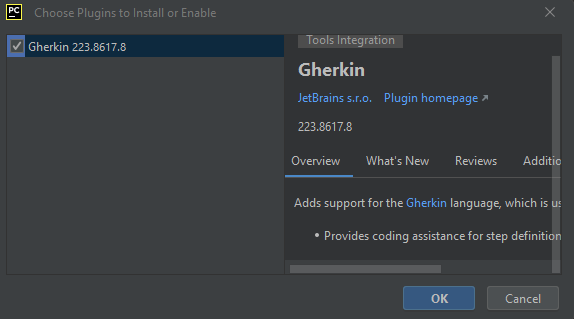
In addition, involving users in the testing process brings a sense of confidence and satisfaction to clients. Their participation ensures that the product aligns with their expectations and needs, instilling trust in its reliability. This testing approach caters to the user's perspective, making it easier for them to articulate their requirements and preferences.

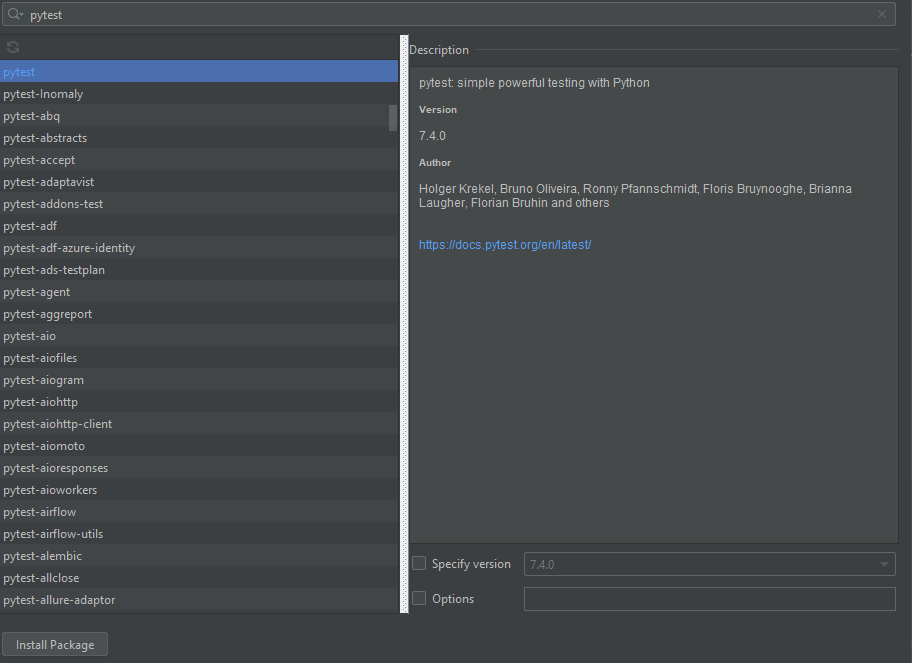
Acceptance testing primarily focuses on Black-Box testing, comprehensively examining the product's entire functionality. This comprehensive testing approach ensures that the product is thoroughly assessed from an end-user perspective.

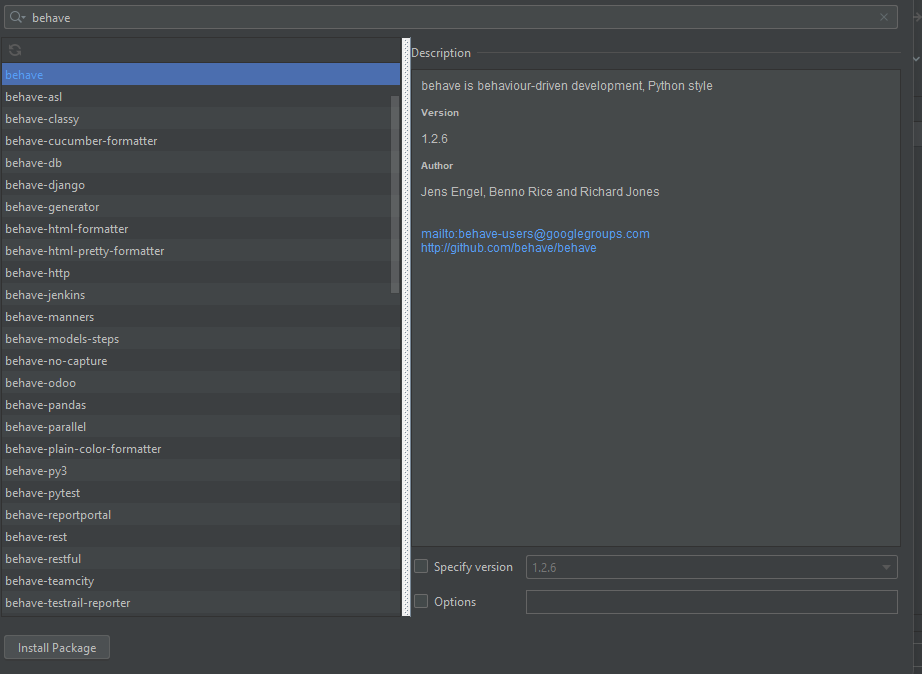
However, acceptance testing does come with certain disadvantages. Users participating in the testing process should possess a basic understanding of the product or application, which might not always be the case. Occasionally, users might be unwilling to engage in the testing process, possibly due to time constraints or other reasons.

Moreover, the feedback collection process can be time-consuming due to the involvement of multiple users. Different opinions and varying perspectives among users can lead to delays in gathering and synthesizing feedback. Unlike other testing phases, the development team is not actively engaged in acceptance testing, which might result in a potential gap between the development and testing phases.

The first thing we must do is to install the Gherkin plugin. the pytest and the behave library.







After that we have to create a program in Python for a TODO\_LIST in which we are going to prove the functionalities of an acceptance test.

The code for the program in python is the following:

**task.py**

class Task:  
 def \_\_init\_\_(self, description, due\_date, priority, completed=False):  
 self.\_description = description  
 self.\_due\_date = due\_date  
 self.\_priority = priority  
 self.\_completed = completed  
  
 @property  
 def description(self):  
 return self.\_description  
  
 @property  
 def due\_date(self):  
 return self.\_due\_date  
  
 @property  
 def priority(self):  
 return self.\_priority  
  
 @property  
 def completed(self):  
 return self.\_completed  
  
 @completed.setter  
 def completed(self, value):  
 self.\_completed = value

**task\_controller.py**

from tallerTodo\_list.TallerTodo\_list.task import Task  
  
  
class TaskController:  
 def \_\_init\_\_(self):  
 self.\_tasks = []  
  
 def add\_task(self, description, due\_date, priority):  
 task = Task(description, due\_date, priority)  
 self.\_tasks.append(task)  
  
 def list\_tasks(self):  
 if not self.\_tasks:  
 print("No tasks found.")  
 else:  
 for index, task in enumerate(self.\_tasks, start=1):  
 status = "Completed" if task.completed else "Pending"  
 print(f"{index}. [{status}] {task.description} (Due: {task.due\_date}, Priority: {task.priority})")  
  
 def mark\_completed(self, task\_index):  
 if 0 < task\_index <= len(self.\_tasks):  
 task = self.\_tasks[task\_index - 1]  
 task.completed = True  
 else:  
 print("Invalid task index.")  
  
 def clear\_tasks(self):  
 self.\_tasks = []  
  
 def sort\_tasks\_by\_priority(self):  
 self.\_tasks.sort(key=lambda task: task.priority)

**app.py**

from task\_controller import TaskController  
  
class ToDoListApp:  
 def \_\_init\_\_(self):  
 self.\_controller = TaskController()  
  
 def run(self):  
 while True:  
 print("\nTo-Do List Manager")  
 print("1. Add Task")  
 print("2. List Tasks")  
 print("3. Mark Task as Completed")  
 print("4. Clear All Tasks")  
 print("5. Sort Tasks by Priority")  
 print("6. Exit")  
  
 choice = input("Enter your choice: ")  
  
 if choice == "1":  
 description = input("Enter task description: ")  
 due\_date = input("Enter due date: ")  
 priority = input("Enter priority: ")  
 self.\_controller.add\_task(description, due\_date, priority)  
 print("Task added successfully.")  
 elif choice == "2":  
 self.\_controller.list\_tasks()  
 elif choice == "3":  
 try:  
 task\_index = int(input("Enter task index to mark as completed: "))  
 self.\_controller.mark\_completed(task\_index)  
 print("Task marked as completed.")  
 except ValueError:  
 print("Invalid task index. Please enter a valid number.")  
 elif choice == "4":  
 self.\_controller.clear\_tasks()  
 print("All tasks cleared.")  
 elif choice == "5":  
 self.\_controller.sort\_tasks\_by\_priority()  
 print("Tasks sorted by priority.")  
 elif choice == "6":  
 break  
 else:  
 print("Invalid choice. Please choose a valid option.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = ToDoListApp()  
 app.run()

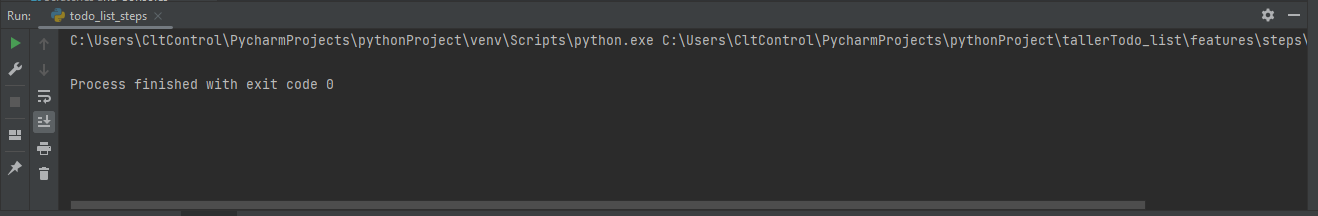
Then we have to create a .feature file to add the feature and the scenarios we are going to test.

Feature: To-Do List Manager  
  
 Suggested Features:  
 • Add a task to the to-do list.  
 • List all tasks in the to-do list.  
 • Mark a task as completed.  
 • Clear the entire to-do list.  
  
 Background:  
 Given the to-do list is empty  
  
 Scenario: Add a task to the to-do list  
 When the user adds a task "Buy groceries"  
 Then the to-do list should contain "Buy groceries"  
  
 Scenario: List all tasks in the to-do list  
 Given the to-do list contains tasks:  
 | Task |  
 | Buy groceries |  
 | Pay bills |  
 When the user lists all tasks  
 Then the output should contain:  
 """  
 Tasks:  
 - Buy groceries  
 - Pay bills  
 """  
  
 Scenario: Mark a task as completed  
 Given the to-do list contains tasks:  
 | Task | Status |  
 | Buy groceries | Pending |  
 When the user marks task "Buy groceries" as completed  
 Then the to-do list should show task "Buy groceries" as completed  
  
 Scenario: Clear the entire to-do list  
 Given the to-do list contains tasks:  
 | Task |  
 | Buy groceries |  
 | Pay bills |  
 When the user clears the to-do list  
 Then the to-do list should be empty  
  
 Scenario: Mark a non-existent task as completed  
 Given the to-do list contains tasks:  
 | Task | Status |  
 | Buy groceries | Pending |  
 When the user attempts to mark a task "Go to the gym" as completed  
 Then the to-do list should remain unchanged

Having created the features we have to write the steps we have to follow to complete that given scenario in a .py. that steps are going to be the acceptance test for the program.

import sys  
from behave import given, when, then  
from io import StringIO  
  
# Create a shared context for scenarios  
from tallerTodo\_list.TallerTodo\_list.task\_controller import TaskController  
  
  
@given("the to-do list is empty")  
def step\_given\_empty\_todo\_list(context):  
 context.controller = TaskController()  
  
@given("the to-do list contains tasks:")  
def step\_given\_todo\_list\_with\_tasks(context):  
 context.controller = TaskController()  
 for row in context.table:  
 description = row['Task']  
 context.controller.add\_task(description, "", "")  
  
@when('the user adds a task "{task\_description}"')  
def step\_when\_add\_task(context, task\_description):  
 context.controller.add\_task(task\_description, "", "")  
  
@when("the user lists all tasks")  
def step\_when\_list\_tasks(context):  
 context.output = []  
 with capture\_stdout() as stdout:  
 context.controller.list\_tasks()  
 context.output = stdout.getvalue().splitlines()  
  
@when('the user marks task "{task\_description}" as completed')  
def step\_when\_mark\_task\_completed(context, task\_description):  
 tasks = context.controller.\_tasks # Not ideal, but for testing purposes  
 for task in tasks:  
 if task.description == task\_description:  
 task.completed = True  
 break  
  
@when("the user clears the to-do list")  
def step\_when\_clear\_tasks(context):  
 context.controller.clear\_tasks()  
  
@then('the to-do list should contain "{task\_description}"')  
def step\_then\_todo\_list\_contains\_task(context, task\_description):  
 tasks = [task.description for task in context.controller.\_tasks] # Not ideal, but for testing purposes  
 assert task\_description in tasks  
  
@then('the output should contain:')  
def step\_then\_output\_contains(context):  
 expected\_output\_lines = context.text.splitlines()  
 assert context.output == expected\_output\_lines  
  
@then('the to-do list should show task "{task\_description}" as completed')  
def step\_then\_task\_marked\_as\_completed(context, task\_description):  
 tasks = context.controller.\_tasks # Not ideal, but for testing purposes  
 for task in tasks:  
 if task.description == task\_description:  
 assert task.completed is True  
 break  
  
@then("the to-do list should be empty")  
def step\_then\_todo\_list\_empty(context):  
 tasks = context.controller.\_tasks # Not ideal, but for testing purposes  
 assert len(tasks) == 0  
  
class capture\_stdout:  
 def \_\_enter\_\_(self):  
 import sys  
 self.\_stdout = sys.stdout  
 sys.stdout = self.\_stringio = StringIO()  
 return self.\_stringio  
  
 def \_\_exit\_\_(self, \*args):  
 sys.stdout = self.\_stdout

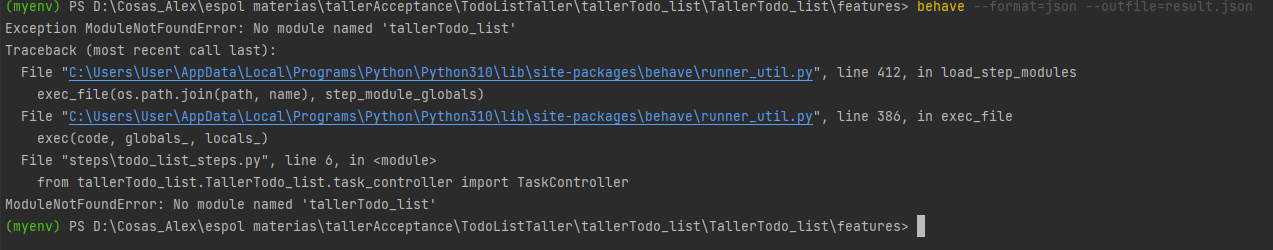
Then, we proceed to run the .py file where the tests are created.



Finally, we have to run the acceptance test and analyze the results with the command behave --format=json --outfile=result.json

In my case, I can not run properly the command because, it appears and error of importing that says that a module was not found or didn’t exist.

I was trying to fix that but, I can’t do it. That’s why there is no result.json file.



In conclusion, acceptance testing is a critical stage in software development that helps ensure the product's quality, alignment with user requirements, and optimal performance. While it brings numerous advantages such as user collaboration, confidence-building, and comprehensive testing, its disadvantages include user knowledge limitations, potential reluctance to participate, and challenges associated with feedback collection and integration.

# Recommendations

1. **Clear Requirements:** Document and agree upon project requirements to guide testing.
2. **User Involvement:** Engage users and stakeholders early in the testing process.
3. **Acceptance Criteria:** Define clear criteria for determining product acceptance.
4. **Test Planning:** Develop a comprehensive testing plan.
5. **User Scenarios:** Create realistic scenarios representing various use cases.
6. **Test Data:** Prepare relevant and realistic test data.
7. **Documentation:** Provide users with clear testing instructions.
8. **Feedback:** Establish a structured feedback mechanism.
9. **User Training:** Train users for accurate test execution.
10. **Automation:** Consider automating repetitive or complex tests.
11. **Diverse Users:** Involve users with different roles and perspectives.
12. **Real-World Conditions:** Test under conditions resembling real-world use.
13. **Feedback Validation:** Validate and prioritize user feedback.
14. **Collaboration:** Foster open communication between teams.
15. **Iterative Approach:** Treat acceptance testing as an iterative process.
16. **Exit Criteria:** Define completion criteria for acceptance testing.

# References:

<https://www.geeksforgeeks.org/acceptance-testing-software-testing/>

# Resources:

Link of the repository: <https://github.com/ajvilleg10/TodoListTaller>