



ATDD Getting Started Guide

This is a very simple, very crude quick start guide to help ASD-200-CMP participants that are new to Robot Framework using Python. This is juuuuust enough to get you started. For real help with syntax and structure, refer to the Robot documentation at <https://robotframework.org/>

None of this code is perfect or ideal. It likely still needs cleanup or refactoring before you're "done". It is, hopefully, enough to get you started.

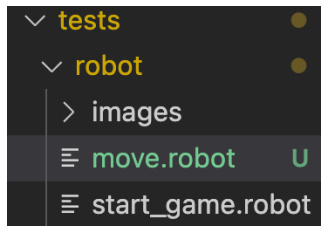
Python with Robot

Robot uses two types of files: 1) .robot and 2) Library. The robot file is a simple definition of what scenarios are being tested and what the test data is. The Library file works like glue - taking the inputs from the Robot file and passing them along to your system operations, and calling assertions to check if the tests pass.

We already have a simple Robot file and Library file in our project (two examples actually!). We COULD add more scenarios to it. But, let's create a separate set of files for move, since it's a distinct system operation.

Robot File

Navigate to tests/robot and create a new file named "move.robot"



Open the file so we can add some content! Let's start with a description of the Feature. Add the following text. Feel free to customize your description.



Now let's add our test data, using what we learned in Specification by Example (SbE). We'll express the Spec by Example table columns. Then, we'll add the examples to the table. Each row can have a descriptive name that helps express what scenario that line is testing.

Add the table. Be sure to use appropriate white space (**two tabs after *** Test Cases *** and one tab between each column**). Note, you may need to adjust or simplify your SbE to transform it into tests. For instance, I made the decision below to separate the X and Y coordinates into separate columns rather than express them as (x,y).

*** Test Cases ***	StartingX	StartingY	Direction	EndingX	EndingY
Move in middle of board	0	0	NORTH	0	1
Move on edge of board	0	0	SOUTH	0	0

Now, create a Keywords section that will define the method signatures for each column in the table. The keywords here should match the column names in your test data, but starting with lower case letters. For each column, define what the method should be called use it. For pre-conditions, call the method something like “initialize with...” or “set...”. For system operations, call them what the system operation is (like move in direction). For post-conditions, name them “...should be”. **Make sure there are tabs before all of your \${var}s.**

```
*** Keywords ***
Move character
    [Arguments]    ${startingX}    ${startingY}    ${direction}    ${endingX}    ${endingY}
    Initialize character xposition with    ${startingX}
    Initialize character yposition with    ${startingY}
    Move in direction    ${direction}
    Character xposition should be    ${endingX}
    Character yposition should be    ${endingY}
```

That’s all we need for our starting robot file! You might need to add move Count or other pre and post-conditions, depending in your requirements. To reach your Definition of Done, you’ll want to go back later and add images and context so this becomes living documentation. To add images, you can adjust the Documentation section to include a URL to an image file.

For EXAMPLE:

```
*** Settings ***
Documentation
...    Example test case using the data-driven (table) syntax.
...    http://arcbotics.com/wp-content/uploads/2015/12/sparki-driver-icon.png
...    You, 1 second ago • Uncommitted changes
```

Will show the image at the URL above in log.html for the test suite.

Library File

The robot file is in human readable syntax and lets you express your test cases. The Library File is there to glue those test cases to your code. We’ll be coding the library file in the language we’re using for our application, in this case Python.

Create a new file in tests/robot/MoveLibrary.py

We'll need 2 imports and 3 attributes to help us store data and call system operations in the GameController class. Inside the MoveLibrary, add the following:

```
from levelup.controller import GameController
from levelup.controller import Direction
```

```
class MoveLibrary:
    start_x: int
    start_y: int
```

```
    controller = GameController()
```

Preconditions

Let's start with our preconditions. The names of these methods should match exactly the keywords you defined in the robot file.

Add the following code directly after the attributes above.

```
def initialize_character_xposition_with(self, x_position):
    self.start_x = x_position
```

```
def initialize_character_yposition_with(self, y_position):
    self.start_y = y_position
```

That's easy enough. The method names maps directly to the keywords in the robot file. We've accepted a parameter input, which will receive the data from the table in the robot file. In this case, we'll take in integers for the X and Y coordinates. Inside the methods, we're storing the inputs so we can use them in a moment.

System Operation

Time to add our system operation call

```
def move_in_direction(self, direction):
    self.controller.set_character_position((self.start_x, self.start_y))
    self.controller.move(Direction[direction])
```

This method is in charge of calling our system operation. It calls a method called "setCharacterPosition". This method isn't a system operation, but it IS a new method we're going to add to allow us to make the GameController testable. With this method, we can now set state, providing a position for the player to start on before a move. Then, we call the move system operation, get the updated game status, and parse out a position from that games

status. If you've just added this code, it won't compile. We need to add the `setCharacterPosition` method to the `GameController` and the `currentPosition` to the `gameStatus` class. Let's do that now!

Updating GameController so we can compile

Open the `GameController` at `src/levelup/controller.py`

First, add a new attribute to the `GameStatus` data class. This will allow us to return the player's current position at any point in the game. For simplicity, I'm using a `Tuple`. You may decide to use another data structure.

```
@dataclass
class GameStatus:
    running: bool = False
    character_name: str = "Character"
    current_position: tuple = (-100, -100)
```

Now, let's add the `setCharacterPosition` method inside the `GameStatus` class:

```
def set_character_position(self, xycoordinates: tuple) -> None:
    print("Set character position state for testing")
    # TODO: IMPLEMENT THIS
```

Wait...why is this just an empty method? Why can't we just add the implementation now? Well...because we're doing the OUTER loop of ATDD right now. We're doing just enough to get the test cases setup and compiling. THEN (later in the course), we'll start implementing methods using Unit Test Driven Development. So, create the empty method, which is just enough to get the code to compile, and leave a `TODO` to implement later. Head back to your `MoveLibrary.py` class and see if you've resolved the compilation errors.

Post-Conditions - MoveLibrary.py

What good is a test without some assertions? The post-conditions are where we put the actual tests for our scenarios. Add the two methods, as shown below to `MoveLibrary.py`

```
def character_xposition_should_be(self, expected):
    end_x = self.controller.status.current_position[0]
    assert end_x == expected, f"Expected: {expected}, Actual: {end_x}"
```

```
def character_yposition_should_be(self, expected):
    end_y = self.controller.status.current_position[1]
    assert end_y == expected, f"Expected: {expected}, Actual: {end_y}"
```

Yay! Failing Tests!

Make sure everything is saved, then rerun `make test-acceptance`

You should see a BUILD FAILED. That's good! We want failing tests, but ones that are wired correctly. If things are working, you should see your new tests failing.

```
=====
Robot | FAIL |
3 tests, 1 passed, 2 failed
=====
Output: /Users/erinperry/dev/python-robot-reference/test_results/robot/output.xml
Log:    /Users/erinperry/dev/python-robot-reference/test_results/robot/log.html
Report: /Users/erinperry/dev/python-robot-reference/test_results/robot/report.html
make: *** [test-acceptance] Error 2
```

Next:

Now it's time to finish up by adding all your test cases to this scenario, as well as create new tests and scenarios for any other acceptance tests you need to add. **You'll need to add MoveCount to your scenarios too, to match your Spec by Example table.** Don't forget to add images, write-ups, etc to make this fantastic documentation too!