Bots (a dancy way to say “testing”)

Juan Cabral - jbc.develop@gmail.com

Jan, 2018

Bots

- You can write “bots” that simulate participants playing your app, so that you can test that it functions properly.
- A lot of oTree users skip writing bots because they think it’s complicated or because they are too busy with writing the code for their app.

Bots

- But bots are possibly the easiest part of oTree. For many apps, writing the bot just takes a few minutes; you just need to write one yield statement for each page in the app, like this:

```
class PlayerBot(Bot):

    def play_round(self):
        yield (views.Contribute, {'contribution': 10})
        yield (views.Results)
```

- Then, each time you make a change to your app, you can run bots automatically, rather than repetitively clicking through.
- This will save you much more time than it initially took to write the bot.
- Also, you can run dozens of bots simultaneously, to test that your game works properly even under heavy traffic and with different inputs from users, preventing any surprises on the day of the study.

Running tests

- Let's say you want to test the session config named **`ultimatum`** in **`settings.py`**.
- To test, open your terminal and run the following command from your project's root directory:

```
$ otree test ultimatum
```

- This command will test the session, with the number of participants specified in **`num_demo_participants`** in **`settings.py`**.
- To run tests for all sessions in **`settings.py`**, run:

```
$ otree test
```

Running tests

Exporting data

- Use the **`--export`** flag to export the data generated by the bots to a CSV file:

```
$ otree test ultimatum --export
```

- This will put the CSV in a folder whose name is autogenerated.
- To specify the folder name, do:

```
$ otree test ultimatum --export=myfolder
```

Writing tests

Submitting pages

- Tests are contained in your app's **`tests.py`**.
- Fill out the **`play_round()`** method of your **`PlayerBot`**.
- It should simulate each page submission. For example:

```
class PlayerBot(Bot):
    def play_round(self):
        yield (views.Start)
        yield (views.Offer, {'offer_amount': 50})
```

- Here, we first submit the Start page, which does not contain a form.
- The next page is Offer, which contains a form whose field is called **`offer_amount`**, which we set to 50.

Writing tests

Submitting pages

- We use **yield**, because in Python, yield means to produce or generate a value.
- You could think of the bot as a machine that yields (i.e. generates) submissions.
- The test system will raise an error if the bot submits invalid input for a page, or if it submits pages in the wrong order.

Writing tests

Submitting pages

- Rather than programming many separate bots, you program one bot that can play any variation of the game, using if statements.
- For example, here is how you can make a bot that can play either as player 1 or player 2.

```
if self.player.id_in_group == 1:
    yield (views.Offer, {'offer': 30})
else:
    yield (views.Accept, {'offer_accepted': True})
```

- Your **if** statements can depend on **self.player**, **self.group**, **self.subsession**, etc.
- You should ignore wait pages when writing bots.

Writing tests

Asserts

- You can use **assert** statements to ensure that your code is working properly.

```
class PlayerBot(Bot):

    def play_round(self):
        assert self.player.money_left == c(10)
        yield (views.Contribute,
               {'contribution': c(1)})
        assert self.player.money_left == c(9)
        yield (views.Results)
```

- **assert** statements are used to check statements that should hold true.
- If the asserted condition is wrong, an error will be raised.

Writing tests

Asserts

- The **`assert`** statements are executed immediately before submitting the following page.
- For example, let's imagine the page_sequence for the game in the above example is **`[Contribute, ResultsWaitPage, Results]`**.
    1. The bot submits **`views.Contribution`**, is redirected to the wait page, and is then redirected to the Results page.
    2. At that point, the **`Results`** page is displayed, and then the line assert **`self.player.money_left == c(9)`** is executed.
    3. If the assert passes, then the user will submit the **`Results`** page.

Writing tests

Testing form validation

- If you use form validation, you should test that your app is correctly rejecting invalid input from the user, by using **`SubmissionMustFail()`**.
- For example, let's say you have this page:

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['int1', 'int2', 'int3']

    def error_message(self, values):
        if values["int1"] + values["int2"] + values["int3"]
            return 'The numbers must add up to 100'
```

Writing tests

Testing form validation

You can test that it is working properly with a bot that does this:

```
from . import views
from otree.api import Bot, SubmissionMustFail

class PlayerBot(Bot):

    def play_round(self):
        yield SubmissionMustFail(views.MyPage,
            {'int1': 0, 'int2': 0, 'int3': 0})
        yield SubmissionMustFail(views.MyPage,
            {'int1': 101, 'int2': 0, 'int3': 0})
        yield (views.MyPage,
            {'int1': 99, 'int2': 1, 'int3': 0})
        ...
```

References

- http://otree.readthedocs.io/en/latest/
- https://docs.python.org/3.6/library/pdb.html
- https://docs.djangoproject.com/en/1.11/topics/i18n/