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:

```python
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})
        ...
```

Writing tests

Test cases

- You can define an attribute cases on your **`PlayerBot`** class that lists different test cases.
- For example, in a public goods game, you may want to test 3 scenarios:
  1. All players contribute half their endowment
  2. All players contribute nothing
  3. All players contribute their entire endowment (100 points)
- We can call these 3 test cases “basic”, “min”, and “max”, respectively, and put them in cases. Then, oTree will execute the bot 3 times, once for each test case. ç
- Each time, a different value from cases will be assigned to **`self.case`** in the bot, so you can have conditional logic that plays the game differently. (see: **`player_bot_example.py`**)

Writing tests

Test cases

- cases needs to be a list, but it can contain any data type, such as strings, integers, or even dictionaries.
- Here is a trust game bot that uses dictionaries as cases. (see: **`player_bot_example2.py`**)

Writing tests

Checking the HTML

- In the bot, **`self.html`** will be a string containing the HTML of the page you are about to submit.
- So, you can do **`assert`** statements to ensure that the HTML does or does not contain some specific substring.
- Linebreaks and extra spaces are ignored.
- **`self.html`** is updated with the next page's HTML, after every **`yield`** statement.
- For example, here is a "beauty contest" game bot that ensures that results are reported correctly: **`check_html.py`**

Writing tests

Automatic HTML checks

- Before the bot submits a page, oTree ensures that any form fields the bot is trying to submit are actually found in the page's HTML, and that there is a submit button on the page. Otherwise, an error will be raised.
- However, these checks may not always work, because they are limited to scanning the page's static HTML on the server side, whereas maybe your page uses JavaScript to dynamically add a form field or submit the form.

Writing tests

Automatic HTML checks

- In these cases, you should disable the HTML check by using **`Submission`** with **`check_html=False`**.

```
from otree.api import Submission

class PlayerBot(Bot)
    def play_round(self):
        yield Submission(
            views.MyPage, {'foo': 99},
            check_html=False)
```

- If many of your pages incorrectly fail the static HTML checks, you can bypass these checks globally by setting **`BOTS_CHECK_HTML = False`** in **`settings.py`**.

Writing tests

Testing timeouts

- You can simulate a timeout on a page by using **`Submission`** with **`timeout_happened=True`**.:

```
from otree.api import Submission

class PlayerBot(Bot)
    def play_round(self):
        yield Submission(
            views.MyPage, {'foo': 99},
            timeout_happened=True)
```

Browser bots

- Bots can run in the browser.
- They run the same way as command-line bots, by executing the submits in your `tests.py`.
- The advantage is that they test the app in a more full and realistic way, because they use a real web browser, rather than the simulated command-line browser.
- Also, while it's playing you can briefly see each page and notice if there are visual errors.

Browser bots

Basic use

- Make sure you have programmed a bot in your tests.py as described above.
- In settings.py, set **`'use_browser_bots': True`** for your session config(s).
- Run your server and create a session. The pages will auto-play with browser bots, once the start links are opened.

Browser bots

Command-line browser bots (running locally)

- For more automated testing, you can use the otree browser_bots command, which launches browser bots from the command line.
- Make sure **Google Chrome** is installed, or set `BROWSER_COMMAND` in **`settings.py`**
- Run your server (e.g. `otree runserver`)
- Close all Chrome windows.
- Run this (substituting the name of your session config):

```
$ otree browser_bots public_goods
```

- This should automatically launch several Chrome tabs, which will play the game very quickly. When finished, the tabs will close, and you will see a report in your terminal window of how long it took.

Browser bots

Command-line browser bots: tips & tricks

- If the server is running on a host/port other than the usual `http://127.0.0.1:8000`, you need to pass `--server-url`

```
$ otree browser_bots public_goods \
    --server-url=http://oTree.herokuapp.com
```

- You will get the best performance if you use **PostgreSQL** or **MySQL** rather than **SQLite**, and use `runprodserver` rather than `runserver`.
- To make the bots run more quickly, disable most/all add-ons, especially ad-blockers. Or create a fresh Chrome profile that you use just for browser testing. When oTree launches Chrome, it should use the last profile you had open.

Browser bots

Choosing session configs and sizes

- You can specify the number of participants:

```
$ otree browser_bots ultimatum 6
```

- To test all session configs, just run this:

```
$ otree browser_bots
```

- It defaults to num_demo_participants (not num_bots).

Browser bots

Browser bots: misc notes

- You can use a browser other than Chrome by setting **`BROWSER_COMMAND`** in **`settings.py`**.
- Then, oTree will open the browser by doing something like

`python subprocess.Popen(settings.BROWSER_COMMAND)`.

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/