Willow, a Python framework for experimental economics

February 4, 2010

Willow, a Python framework for experimental economics

Contents

1 About this manual

You are reading the manual for Willow, a Python framework for experimental economics. This manual presumes that you already have an understanding of experimental economics, of HTML, and of Python programming.

If you are interested in learning about experimental economics, you could do worse than to have a look at the following textbooks:

- Daniel Friedman, *Experimental Methods: A Primer for Economists*
- John Kagel and Alvin Roth, *The Handbook of Experimental Economics*

In order to get started with Python if you already know how to program in some other language, you can try

- Mark Pilgrim, *Dive into Python*

  `http://diveintopython.org`

- Guido van Rossum et al., *The Python Tutorial*

  `http://docs.python.org/tutorial/`

If you want to learn both the craft of computer programming and the Python programming language at the same time, you are in luck, since Python is an excellent first language. Some free resources include:

- Jeffrey Elkner, Allen B. Downey, and Chris Meyers, *How to Think Like a Computer Scientist*

  `http://openbookproject.net/thinkCSpy/`

- Swaroop C. H., *A Byte of Python*

  `http://www.swaroopch.com/notes/Python`

2 Why Willow?

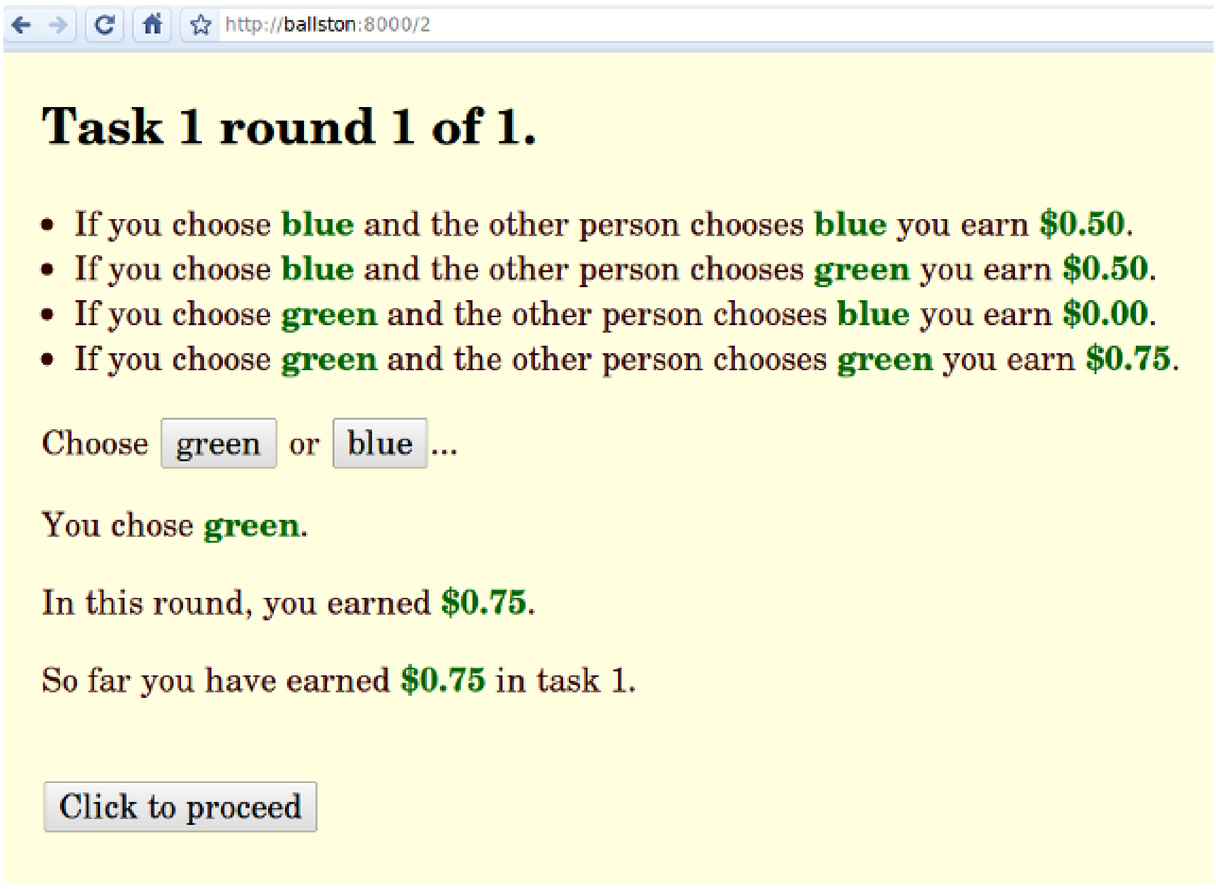
Willow is a tool for making the kinds of computer user interfaces typically used in experimental economics. Without Willow, you might design such an interface from scratch, or by modifying earlier from-scratch software written for similar experiments. Using Willow, you can do the same thing, but faster and better.

Willow sets out to accomplish this goal not by providing canned routines for commonly performed experiments. After all, it is precisely because our experiments are different from what has gone before that we bother with them. Rather, Willow is a generic toolkit that makes a lot of the things we do in experimental economics easy.

To use a Willow program, you need only install your program on a single "monitor" computer. The computers used by your research subjects will simply run a web browser, which connects to a web server that runs on the monitor computer. The user interface for the subjects will be displayed inside the browser window.

3 Screenshots

Willow is a versatile program, and you can use it for many different interfaces; but you might just be curious what a Willow interface can look like. Here are some examples taken from the first study that used Willow at the Center for the Study of Neuroeconomics at George Mason University.



Task 1 round 1 of 1.

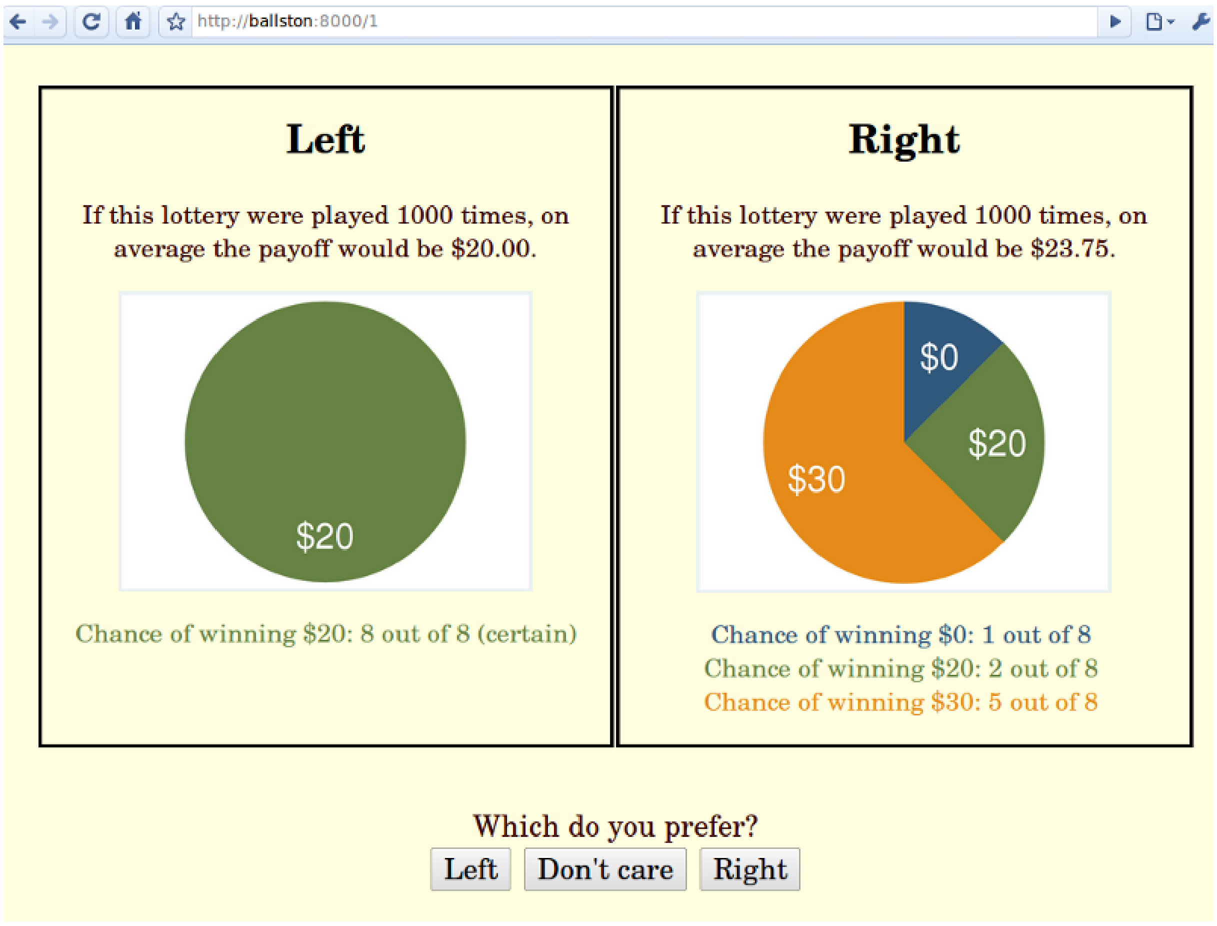
- If you choose **blue** and the other person chooses **blue** you earn **$0.50**.
- If you choose **blue** and the other person chooses **green** you earn **$0.50**.
- If you choose **green** and the other person chooses **blue** you earn **$0.00**.
- If you choose **green** and the other person chooses **green** you earn **$0.75**.

Choose green or blue ...

You chose **green**.

In this round, you earned **$0.75**.

So far you have earned **$0.75** in task 1.



Left

If this lottery were played 1000 times, on average the payoff would be $20.00.

Chance of winning $20: 8 out of 8 (certain)

Right

If this lottery were played 1000 times, on average the payoff would be $23.75.

Chance of winning $0: 1 out of 8
Chance of winning $20: 2 out of 8
Chance of winning $30: 5 out of 8

Which do you prefer?

4 Installing Willow

To install Willow, you extract the zip file to a convenient location. You will do this anew for each of your Willow projects. In addition, you will need to have Python installed. Willow has been tested using Python 2.6 on Ubuntu GNU/Linux. You should be able to use it with Python 2.6 on Windows and Mac OSX as well. If you do not have Python 2.6 on your computer, get it from http://www.python.org/download/releases/2.6.3/ and then come back. If you are not sure whether you have Python 2.6 on your computer, just move on to the next section of the manual. If things don't work, you'll notice soon enough, and you can go back and install Python. Do not use Python version 3. It will not work.

5 Using Willow

In the Willow folder, you will find a file named `hello.py`. You need to run this file. On Windows and most Linux distributions, you can double click on it. You can also run the command `python hello.py` from the command line. Once the `hello.py` program is running, fire up a web browser and point it to `http://localhost:8000`. You should see a page that looks like so:

Hello World.

In practice, while programming, you will want to run Willow from within IDLE, which is the integrated development environment that comes with Python. To do so, right-click `hello.py` and select "Edit with IDLE". This will pop up an editor window with the `hello.py` code in it:

```python
from willow.willow import *

def session():
  add("<h1>Hello, world.</h1>")

run(session)
```

See here the code of a very simple but working Willow program. You can run it using the `Run > Run Module` menu option or by pressing `F5`. Again, you now need to point a web browser to `http://localhost:8000`.

If you see an error message like "address already in use," you are probably trying to run two instances of Willow simultaneously. You should stop the first one first (just close the window).

Now, let us look at the above code. Most of it is boilerplate, but the good news is that this is all the boilerplate you need. The first line imports the willow library. It assumes that `willow.py` is in a subdirectory `willow` of the directory in which `hello.py` is; this is how things are set up for the sample programs, and if you put your own programs in the same place, it will just work. The next two lines define a session function, and the last line instructs Willow to start the web server, using the session function that has just been defined.

Together, these four lines define a web server that you can connect to from a browser. If the browser is running on the same machine as the server, you can use `http://localhost:8000` as a URL. If it is running somewhere else, you must find out the IP address of the server. (If you don't know it, there is a whole chapter further down in the manual about how to figure it out.) If the IP address is, say, 123.123.123.123, then you can reach your Willow server using `http://123.123.123.123:8000`. The `:8000` part is called the port number; you can specify a different port number as an extra argument to `run` (e.g. `run(session, 8001)`) if you insist on a port different from 8000.

Once the web server is running, for each client that connects to it, the session function will be called once. It will keep running until it returns. Each session function runs in its own thread. This means that several "copies" of the session function are running simultaneously.

In this case, we want to simply display the text "Hello world." in the web browser of every client that connects, so we have a very simple session function that does nothing but `add("<h1>Hello, world.</h1>")`. The `add()` function adds an HTML snippet to the web page being displayed in the client's browser.

6 A digression on your IP address

You need the IP address of your monitor computer in order for your client computers to connect to it. It would be convenient if Willow could display the right IP address for you to use, but unfortunately, this is easier said than done, since a computer can have multiple IP addresses at the same time: IP version 4 and IP version 6, on the wireless network, on the ethernet network, on the "loopback" network, and so on.

6.1 How to find your IP address on Linux

On Linux on my computer, I use the `ifconfig` command at the command line. When I enter

```
$ ifconfig
```

the computer responds

```
$ ifconfig
eth1      Link encap:Ethernet  HWaddr 00:23:ae:1e:9e:ac
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:17

eth2      Link encap:Ethernet  HWaddr 00:24:2b:c6:a4:76
          inet addr:10.143.91.80  Bcast:10.143.91.255  Mask:255.255.254.0
          inet6 addr: fe80::224:2bff:fec6:a476/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10076 errors:0 dropped:0 overruns:0 frame:7868
          TX packets:10906 errors:20 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1571308 (1.5 MB)  TX bytes:1341157 (1.3 MB)
          Interrupt:17 Base address:0xc000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:85540 errors:0 dropped:0 overruns:0 frame:0
          TX packets:85540 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:41237280 (41.2 MB)  TX bytes:41237280 (41.2 MB)
$
```

TIP

In this and many other software manuals, `$` is used to indicate the Linux/UNIX prompt. On your computer, it may be different. `%` is a popular choice, too, for example.

There is much information here, but we only need a little. First, notice that this computer has three network interfaces, called `eth1`, `eth2` and `lo`. `lo` is the loopback device; it does not correspond to any actual hardware, but it is rather a contraption that the operating system uses so that you can make connections to your own machine at address `127.0.0.1` even when the network is down. The other two network devices are actual network cards that are in the computer; as it happens, `eth1` is a wired Ethernet network, and `eth2` is a wireless 802.11g network. According to the information above, the computer has IP address 10.143.91.80 on `eth2`, and it has no IP address on `eth1`, which makes sense, because the wired connection was not plugged when I executed the command. In the output above you can also see IP v6 addresses (labeled `inet6 addr`), but those are not commonly used.

6.2 How to find your IP address on Windows

BLABLABLA

7 A digression on HTML, the hypertext markup language

In order to use Willow, it is necessary that you understand some HTML, because HTML is the language of web pages, and the user interface for a Willow program is made up of web pages. This manual will not go into detail about HTML, but here is a summary of some of the most common HTML tags:

```
<h1>This is a level 1 heading.</h1>
<h2>This is a level 2 heading.</h2>
<h3>This is a level 3 heading.</h3>
<p>This is a paragraph, with <b>bold</b> and <i>italics</i>.
<li>This is a bulleted list item.
<input type="submit" id="foo" value="This is a button">
<input type="text" id="foo" value="This is a text box">
```

8 A more complicated Willow example

BLABLABLA

9 API documentation

NOTE

The `selector` argument can take any CSS selector. The CSS selector mini-language is rather rich, but the most typical cases are things like `p` (to get at all `<p>` elements), `#a`, (to get at the first element with `id="a"`), and `.a` (to get at all elements with `class="a"`). For more information, consult any book or web page on CSS. Whenever `selector` is omitted, the selector "`body`" is implied, which means you are referring to the entire body of the HTML document.

NOTE

By specifying an integer, or a sequence of integers, for `number`, you can operate directly on clients other than the one associated by default with the current session thread.

NOTE

Whenever `argument` is a file descriptor that can be read from, the contents of the file are automatically substituted.

`me()` Returns the number of the current client.

`url()` Returns the pathname part of the URL used to first load the current client. For instance, if the client was loaded as `http://localhost:8000/`, this returns "`/`", and if the client was loaded as `http://localhost:8000/3`, this returns "`/3`".

`set(content, [selector], [number])` Set the content of the HTML element(s) referred to by `selector` on to be `content`.

`add(content, [selector], [number])` Add `content` to the content of the HTML element(s) referred to by `selector`.

`tweak(value, attribute, [selector], [number])` Change the value of attribute `attribute` to be `value` in HTML element(s) referred to by `selector`.

`push(style, [selector], [number])` Add the CSS style `style` to the HTML element(s) referred to by `selector`.

`pop(style, [selector], [number])` Remove the CSS style `style` from HTML element(s) referred to by `selector`.

`hide([selector], [number])` Hide the HTML element(s) referred to by `selector`.

`show([selector], [number])` Show the HTML element(s) referred to by `selector`, if previously hidden.

`peek([selector], [number])` Request the contents of the HTML element(s) referred to by `selector` to be posted as `("peek",number,contents)`. Intended to be used with `<input type=`*`text`*`>` elements.

`hold([number])` Stop executing subsequent `set`, `add`, `push`, `pop`, `hide`, `show`, and `peek` immediately, but instead stuff them in a buffer where they sit until `flush()` is called.

`flush([number])` Execute all `set`, `add`, `push`, `pop`, `hide`, `show`, and `peek` actions that were held in a buffer since `hold()` was called, and from here on out resume executing such actions immediately without buffering.

`put(tuple, [delay])` After an optional `delay` in seconds, post the tuple onto the communication board.

WARNING

In Python, the value `x` is the same as `(x)`, but it is not the same as the one-element tuple containing it, which is written `(x,)`.

`get(*queries)` Block until a tuple becomes present on the board that matches at least one of the queries, then return it. A tuple on the board matches a query whenever the tuple on the board and the query have the same length, and each element of the query is either `None` or identical to the corresponding element in the tuple on the board.

`grab(*queries)` If a tuple is present on the board that matches at least one of the queries return it; otherwise return `None`.

`run(session, [port])` This starts the Willow web server on the specified port (or port 8000 if you leave the second argument out). The web server will create a thread for each client that connects, and each thread will run the `session` function. Within each thread, the `me()` function will return a different value, and `add`, `set`, etc. will by default operate on a different client. The web server will search for requested files in the current directory, among other places, so you can for instance include images in your pages.

`config(filename)` Reads a configuration file in JSON format (http://json.org/) and returns a corresponding Python object. For instance, if you have a file `protocol.txt` containing the text `{ "rounds" : 12, "subjects" : 4}` then `cfg = willow.config("protocol.txt")` will set the variable `cfg` to the python dictionary `{u'rounds': 12, u'subjects': 4}`.

WARNING

The letter `u` appears in front of the strings that come out of `config` because JSON strings are Unicode, not ASCII, but you do not need to worry; to get the value of the `rounds` parameter, simply use `cfg["rounds"]`.

`log(x,y,...)` Write (x,y,...) to the log file as a record. A new log file is created in the `log` folder whenever the Willow library is loaded, and it bears a name based on the date and time when it was created. The log files are in CSV format and can be opened with a text editor or a spreadsheet program.

10 Willow CSS tricks

Some Willow features are accessed by means of CSS classes:

`hidden` Any element that has class `hidden` turned on will start out hidden. You can call `show()` on it to make it appear. Example: `<p>At the moment, you can<span class="hidden">not</span> trade widgets.</p>`

`bait` Any element that has class `bait` turned on will automatically get the additional class `mouse` turned on whenever the mouse pointer is positioned over that element. This can be used to create "hover" effects.

11 Kiosk mode browsing

In order to prevent subjects from fiddling around with the web browser used to display a user interface to them, you can use a "kiosk mode" web browser. While several web browsers have kiosk modes, I have found that Google Chrome is the most convenient. On Linux, you simply use the command line `google-chrome --kiosk` instead of just `google-chrome`, and you get a full-screen browser without any extraneous bits of user interface that subjects might accidentally click on.

12 Architecture

In this section, I will describe some of the nuts and bolts of Willow. It is not usually necessary to understand this, but it may be helpful for those seeking to extend Willow or satisfy their curiosity.

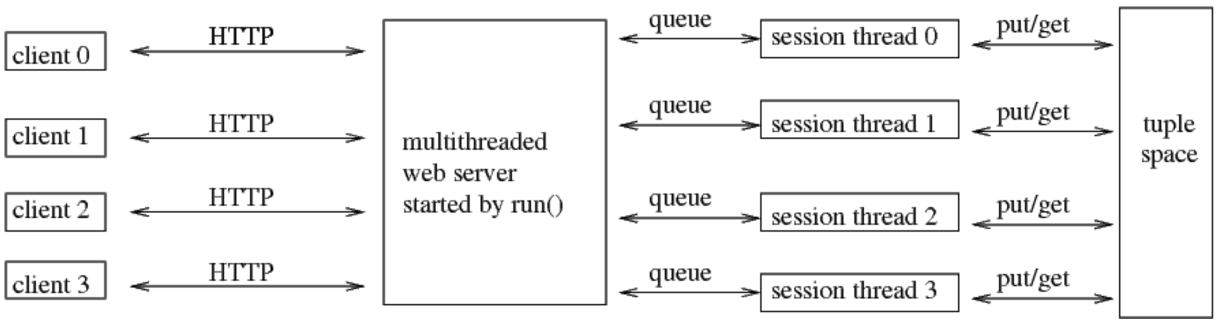
The core of willow is made up of `willow/willow.py` and `willow/willow.js` (which is included by `willow/index.html`). In `willow.py`, the function `run` is defined, which starts a web server. This web server serves GET requests in pretty much the normal way, but it does something special using POST requests. In particular, it expects POST requests to be Ajax calls ("`XMLHttpRequest`") from `willow.js`. In fact, each running instance of `willow.js` (and there is one per client) will always have one such request outstanding; when a response comes in, a new one is generated immediately. This means that the web server always has a way of talking to the client: it can respond to the outstanding POST request. In order to keep track of the outstanding POST requests, the web server is multithreaded; each request gets its own thread. The web server threads, in their turn, decide what to reply to a POST request by looking on a queue.

For each client, there is a separate queue. Each client generates, in JavaScript, a UUID, which it submits when it first connects, and which the web server can use to map a POST request to the correct queue. The reason that we do not use the sequential client numbers for this purpose is that the way the protocol works implies that the first contact is made by the client, and the clients have no way of doing the communication required to make sure their client IDs are sequential before they are actually talking to the server. UUIDs, on the other hand, which are hashes of the wall clock time and a random number, can be generated in a decentralized way. Once clients connect, they are also assigned a sequential client number, which is exposed to the user.

The web server does not busy itself with putting things on the queues. That is what session threads are for. One session thread is started each time a client connects for the first time. The session thread is configurable; in fact, in runs whatever `session` function is passed to `run()`. Each session thread is assigned a name which is the sequential client number, and the queues are made indexable by these numbers as well. (Note that this means we have two different tables pointing to the same queues, one indexed by UUID, and one by client number.)

It is these `session` functions that put things onto the queues. Each session function has access to each of the queues, indexed by client number, but it typically uses functions that automatically fill in the client number that the particular session thread is associated with. This is accomplished, of course, by looking at the thread name of the current thread.

Last, we come to the tuple space. This is largely a convenience, and could be used independently of the rest of Willow, but it has proven to be a convenient way of doing IPC between the threads without needing to understand overly complicated concurrency primitives.



13 Twig

Twig is an extension to Willow. It is a repository of code that shows up over and over in experimental interface programs: code for logging in terminals in a certain order, for doing surveys, etc. All of Twig is written on top of Willow, and there is nothing in it that you couldn't do yourself using Willow and some Python standard library functions. If a Twig functions doesn't fit your needs, you can always write your own (and maybe look at the Twig source code for inspiration.)

To use Twig in your code, you have to add an additional import line to the beginning of your file:

```
from willow.twig import *
```

Twig internally uses tuples and HTML `id+s that start with +`__ (two underscores). If you plan to use Twig, you should avoid using any tuples or HTML +id+s that start with two underscore, so as not to interfere with Twig.

Twig is still under development, and as of right now, the interface to Twig functions is likely change from one Willow version to the next. (The interface to Willow functions may also change, but I try to avoid that, and make mention of it in the manual.)

13.1 `assemble()`

Willow offers only a very basic mechanism for identifying clients: the numbers returned by `me()`, which are assigned to clients in the order in which they first connect. If you have a program launcher in your lab that automatically launches clients on a number of terminals around the room, you can end up with these client numbers being distributed around the room in strange ways, based on which machines are the fastest to connect. In practice, you may want to have a more stable mapping between terminals and client identifiers. That is where `assemble()` comes in.

The way you typically call `assemble()` is at the beginning of your session function, like so:

```
def session():
    number, numbers = assemble()

    # ... your code ...

run(session)
```

Now, when you connect a bunch of clients, you will see a button labeled "0" on the first ("monitor") client, and buttons labeled "Log in" on all other ("subject") clients. As you click these "Log in" buttons on the subject clients, they will each be assigned a number. These numbers are independent from the numbers returned by `me()`, and they are assigned, starting with 1, in the order in which the "Log in" buttons are clicked.

When you have pressed all the "Log in" buttons, you can go back to the monitor client, and you will see that by now, it has a sequence of buttons with labels like:

- 0
- 0, 1
- 0, 1, 2
- 0, 1, 2, 3

These buttons allow you to decide which of the subjectclients to actually use. This is particularly useful when you want to set up an experiment that may have a different number of participants based on actual turnout: you can set up the lab with, say, 12 logged in machines, and if only 10 students turn out, you click the button labeled "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10+.

Once one of the buttons on the monitor is clicked, the `assemble()` function returns a tuple of two elements. The first is the subject ID newly associated with the current thread, or, if the subject is not included on the button clicked on the monitor, `None`. The second is a list of all the subject IDs, including `0` (which refers to the monitor).

One further feature of `assemble()` is revealed when you use non-standard URLs to load the subject clients. For instance, if your Willow instance is running on a machine with IP address `1.2.3.4`, you would normally use `http://1.2.3.4:8000/` to connect, but you could also use `http://1.2.3.-4:8000/42`. In that case, the subject identifier will be assigned not sequentially but according to the number at the end of the URL. This is useful if you have a centralized program launcher facility in your lab that is able to open different web pages on different terminals. In that case, you can simply configure the program launcher to open `http://1.2.3.4:8000/1` on the first machine, `http://1-.2.3.4:8000/2` on the second machine, etc., and you will be assured that subject IDs are assigned in a predetermined way.

Here's an example of how to use `assemble()`:

```
def session():
    number, numbers = assemble()
    if number == 0:
       set("MONITOR<p>Using Subject IDs %r" % numbers)
    else:
       set("<p>This client has Subject ID %d" % number)

run(session)
```

13.2 survey(title, number, questions, [selector])

This is used for surveys. A typical use would be:

```
MYSURVEY = [
  ("Do you like chocolate?", ("Yes", "No"))
]

if me() > 0:
  survey("FOOD", me(), MYSURVEY)
```