Willow: Experimental Economics with Python

1. About this manual

You are reading the manual for Willow, a Python framework for experimental economics developed at the Center for the Study of Neuroeconomics at George Mason University. 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`

Also, in order to use Willow, you need to have basic knowledge of HTML and CSS, the languages used to build web pages. For this I strongly recommend

- Kennedy and Musciano, *HTML & XHTML, The Definitive Guide*, O'Reilly.

O'Reilly books are available full-text through an online service called Safari, which many universities subscribe to. There are also many free on-line sources about HTML and CSS.

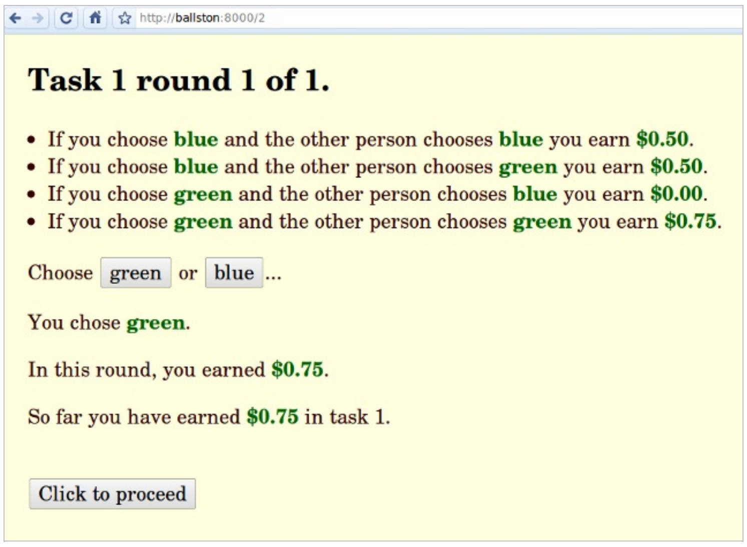
2. Why Willow?

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.

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 and the Interdisciplinary Center for Economic Science at George Mason University.



Task 1 round 1 of 1.

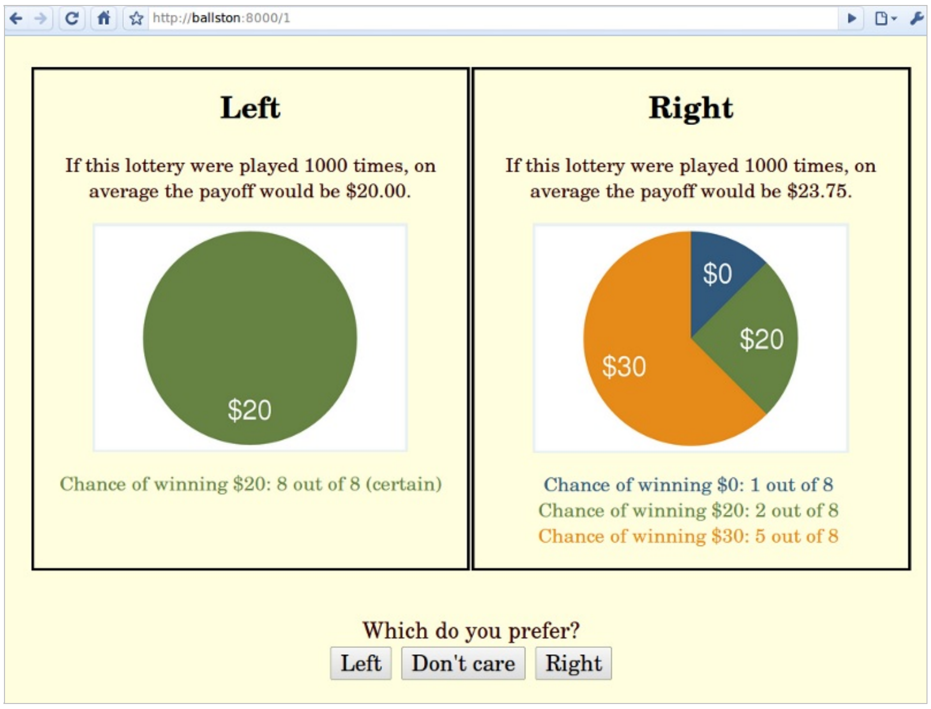
- If you choose **blue** and the other person chooses **blue** you earn **$0.50**.
- If you choose **blue** and the other person chooses **green** you earn **$0.50**.
- If you choose **green** and the other person chooses **blue** you earn **$0.00**.
- If you choose **green** and the other person chooses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

3. Preliminary 1: HTML

Willow relies on HTML, the hypertext markup language, for the purpose of constructing a user interface. A complete description of HTML is outside the scope of this manual, but I'll give a brief introduction.

An HTML document is a text file with some special markup sequences. To practice with HTML, you can construct HTML documents with a text editor, and load them in a web browser.

Consider this example:

```
<p>This is a paragraph, with a <b>bold</b> bit in it, and a <br>line
break and a <a href="http://google.com">link</a>.</p>

<ol>
 <li>This is         the first item of an ordered list.</li>
 <li>This is the second        item of an ordered list.</li>
</ol>
<ul>
 <li>This is the first item of an unordered list. </li>
 <li>This is the second item of an unordered list.</li>
</ul>

<div>This is a div. It has a <span>span</span> in it.</div>
<div>This is a div. It has another <span>span</span> in it.</div>
```

If we type this into a file called `example.html` and load that file in a web browser, this is what we see:

Even though this is a simple example, it already illustrates some of the most important points about HTML:

- HTML is made up of *elements*, which are delimited by *tags*. For instance, the above example starts with a `p` element, the start of which is marked with an opening tag `<p>`, and the end of which is marked with a closing tag `</p>`.
- Elements can have contents. For example, the `b` element in the above example contains the text `bold`.
- Some elements, such as `br` do not have contents, and those do not need a closing tag. (Sometimes, you will see these written like `<br />`; this is an XML thing, which you can read about elsewhere.)
- Elements also can have attributes. In the above example, the `a` element has an `href` attribute, with value `http://google.com`.
- In HTML, any sequence of newline characters and/or blank characters is always treated as a single blank. Extra blanks, or newlines, in the HTML source have no effect. If you do want a line break, you can use the the HTML element `<br>`.
- Most HTML elements have standard meanings for the browser. The `b` element is rendered by typesetting its contents in bold, the `a` element describes a link, etc. There are two exceptions: `div` and `span`. These have no inherent meaning, and only become useful when we start using things like CSS. The difference between `div` and `span` is that a `div` is typeset above/below whatever comes before/after it, while a `span` is typeset in-line.

Next, we will see a list of common HTML elements.

`<p>…</p>`
: A paragraph.

`<ul>…</ul>`
: An unordered (bulleted) list.

`<ol>…</ol>`
: An ordered (numbered) list.

`<li>…</li>`
: An item for a list.

`<div>…</div>`
: A generic chunk of text displayed vertically.

`<input>`
: An input widget

`<b>…</b>`
: A piece of text in bold

`<i>…</i>`
: A piece of text in italics

`<span>…</span>`
: A generic chunk of text displayed in-line.

`<a>…</a>`
: A link

`<br>`
: A line break

`<hr>`
: A horizontal line

`<table>`
: A table

`<tr>`
: A table row

`<td>`
: A table element

The last three of these fit together like so:

```
<table>
<tr><td>Column 1, Row 1</td><td>Column 2, Row 1</td></tr>
<tr><td>Column 1, Row 2</td><td>Column 2, Row 2</td></tr>
</table>
```

Some of the more common HTML attributes:

`<table border=`*`1`*`>…</table>`
: A table with 1 pixel borders.

`<a href="http://google.com/">…</a>`
: A link to `http://google.com/`

`<input type=`*`submit`*` value=`*`foo`*`>`
: A button with label `foo`

`<input type=`*`text`*`>`
: A text field.

That's all for now. References on HTML abound both online and in print.

4. Preliminary 2: CSS

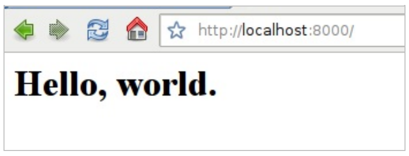
Yet to be written.

5. Lesson 0: Wherein we install 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.

6. Lesson 1: Wherein we meet Willow

In the Willow folder, you will find a file named `lesson.01`. 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 lesson01.py` from the command line. Once the `lesson01.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 `lesson01.py` and select "Edit with IDLE". This will pop up an editor window with the `lesson01.py` code in it:

```
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 `lesson01.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. We say that each session function runs in its own "session thread". This means that several "copies" of the session function are running simultaneously, one for each client.

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.

Try connecting to `localhost:8000` in multiple browser tabs. You'll see that each one shows "Hello, world." This is because the `session()` function gets called once for each client that connects.

7. Lesson 2: Wherein all clients are not equal

In real applications, we usually do not want to do the same for every client that connects. More likely, we'll want one client to show some sort of experimenter console, one client to show the interface for subject 1, one client the interface for subject 2, etc.

Have a look at `lesson02.py`:

```
from willow.willow import *

def session():
  if me() == 0:
      add("Hello monitor %d at URL %s" % (me(), url()))
  else:
      add("Hello subject %d at URL %s" % (me(), url()))

run(session)
```

The `%` operator in Python is very useful for constructing strings. You can read about it in section 6.6.2, "String Formatting Operations", of the Python 2.6.4 Standard Library reference manual.

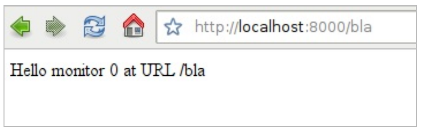
As you can see, there is still only one `session()` function, and for each client that connects, a session thread is started that runs the `session()` function. Each client runs the exact same function, but… inside the `session()` function the value returned by `me()` will be different in each thread. In particular, each client gets a number, starting with 0, in the order in which the clients first connect, and `me()` returns the number of the client associated with the current thread.

The program also makes use of a second mechanism that can be used for distinguishing clients, and that is `url()`. As you may have noticed, Willow is not particularly picky about the URL you use to load it. You could have typed `http://localhost:8000/` or `http://localhost:8000/3` or `http://localhost:8000/spamandtomatoes` and in any of those cases, the same page would show and the same `session()` function would be called; but inside each `session()` function, `url()` would return a different value.

Try running `lesson02.py` and then try to connect to Willow with various URLs. Each URL has to be of the form `http://<the-host-running-willow>:8000/<something>`, where

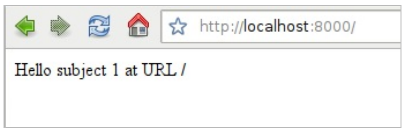
`<the-host-running-willow>` can be `localhost` or your IP address or hostname, and `<something>` can be anything, including the empty string.

You should see pages looking like this:



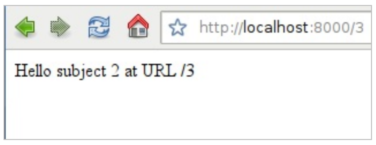
http://localhost:8000/bla

Hello monitor 0 at URL /bla



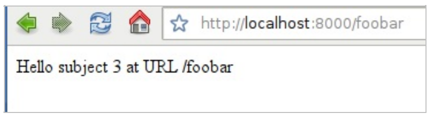
http://localhost:8000/

Hello subject 1 at URL /



http://localhost:8000/3

Hello subject 2 at URL /3



http://localhost:8000/foobar

Hello subject 3 at URL /foobar

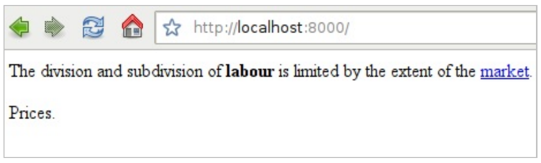
8. Lesson 3: Wherein we produce output

In this lesson, we will see some of the Willow functions for manipulating user interface elements. All of these are based on HTML and CSS, and it is crucial that you acquire a basic understanding of these languages. An extensive description of HTML and CSS is outside the scope of this manual.

Have a look at `lesson03.py`.

```
from willow.willow import *

def session():
    add("<style type='text/css'>.important { font-weight: bold; }</style>")
    add("<p>The <span class='important' id='a'>division</span> of "
        "<span class='b'>labor</span> is limited "
        "by the extent of the "
        "<a href='http://ebay.org'>market</a>.")
    add(" and subdivision", "#a")
    set("labour", ".b")
    tweak("http://ebay.com", "href", "a")
    pop("important","#a")
    push("important",".b")
    add("<p class='elusive'>Microfoundations.</p>")
    add("<p class='hidden'>Prices.</p>")
    hide(".elusive")
    show(".hidden")

run(session)
```

This displays a web page like so:


http://localhost:8000/

The division and subdivision of **labour** is limited by the extent of the market.

Prices.

This shows off all of Willow's UI manipulation functions. Let's go through it line by line.

```
add("<style type='text/css'>.important { font-weight: bold; }</style>")
```

First we add a style sheet to the page. Again, CSS is beyond the scope of this manual and you should go out and read up on it, but what this comes down to is that we instruct the browser to typeset in bold all HTML elements that have class "important" turned on.

```
add("<p>The <span class='important' id='a'>division</span> of "
    "<span class='b'>labor</span> is limited "
    "by the extent of the "
    "<a href='http://ebay.org'>market</a>.")
```

Note that in Python string literal juxtaposition adds the string literals together, which makes for a convenient way of splitting long strings over several lines.

The `<span>` element in HTML has no default meaning, and without any style sheets or other manipulations, a piece of HTML enclosed in a `<span>` is exactly identical to the same piece of HTML not so enclosed. In the above fragment, we have two `<span>` elementss. Note that because the style sheet instructed the browser to print all elements of class `important` in bold, the word "division" is printed in bold; or at least, it would be, if we did not manipulate it further.

```
add(" and subdivision", "#a")
```

This shows that you can add text not just to the page as a whole, but also to particular elements. The optional second argument to `add` is a CSS selector. The most common CSS selectors are:

- `p`, which refers to all `<p>` elements;
- `h1`, which refers to all `<h1>` elements;
- etc.;
- `.x`, which refers to all elements with `class=`*x*;
- `#a`, which refers to the first element with `id=`*a*.

Note that some selectors can refer to more than one element, and in that case, your HTML snippet gets added to all matching elements.

In this case, what we are asking Willow to do is to find the first HTML element with `id=`*a* and add the text " and subdivision" to the end of it it.

```
set("labour", ".b")
```

The `set()` function is just like the `add()` function except that instead of adding HTML to the end of some element, it replaces the entire contents of the element. In particular, `set("")` clears the entire page!

```
tweak("http://ebay.com", "href", "a")
```

So far, we have seen the functions `add` and `set` that manipulate the contents of HTML elements. But HTML elements have attributes as well as contents. For instance, if you write `<a href="foo">…</a>`, you have created an `<a>` element with contents "…" and with an `href` attribute with value `foo`. The `tweak()` function serves to manipulate attributes. In this case, we set the `href` attribute of **every single `<a>` element on the page** to be `http://ebay.com`.

```
pop("important","#a")
push("important",".b")
```

In HTML, a single element can have more than one class. You can write that as `<p class=`*`class1 class2`*`>…</p>`. The `pop()` and `push()` functions are there to remove a class from an element and to add a class to an element, respectively, without changing any other classes that the element might have.

```
add("<p class='elusive'>Microfoundations.</p>")
add("<p class='hidden'>Prices.</p>")
hide(".elusive")
show(".hidden")
```

The `hide()` and `show()` functions do just that: hide and show elements. Willow is set up so that any element with class `hidden` set is hidden by default and only shows up when you call `show()` on it.

So far, we have seen the Willow UI primitives: `add`, `set`, `tweak`, `push`, `pop`, `hide`, and `show`. In combination with HTML and CSS, these primitives make a very powerful UI language.

9. Lesson 4: Wherein we chatter

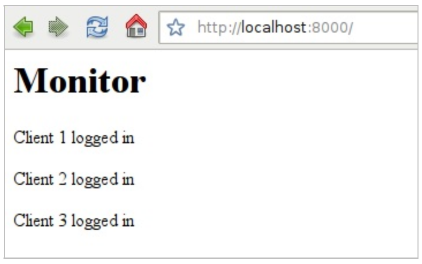
Now that we have seen how to produce output on the client screens, you may be wondering how we can read input. But before we get to that, we will first discuss another important feature of Willow: its communication system.

Consider `lesson04.py`.

```
from willow.willow import *

def session():
  if me() == 0:
    add("<h1>Monitor</h1>")
    while True:
      _, n = get(("HELLO", None))
      add("<p>Client %d logged in" % n)
  else:
    put(("HELLO", me()))
    add(me())

run(session)
```

In this short program, we can see a typical use case for the important Willow functions `put()` and `get()`. The first client, for which `me()==0`, acts as a control panel that displays information about other clients coming online. In order to convey that information, each client other than the first puts a tuple `("HELLO", me())` on the Board. The first client retrieves these tuples and uses them to display information about logins.

After the monitor client and 3 other clients have connected, the monitor client display would look like this:



Monitor

Client 1 logged in

Client 2 logged in

Client 3 logged in

To fully understand this, we must understand the Board. The Board is the one and only data structure that is shared between the various session threads in Willow. The board contains tuples. Tuples are immutable sequences, and they are built into Python. For instance, `(2,3)` is a two-element tuple, and `(2,3,"a")` is a three-element tuple.

One peculiarity of Python is that a one-element tuple is written `(2,)` to distinguish it from `(2)`, which is just another way of writing `2`.

To put *`my_tuple`* on the board, you call `put(`*`my_tuple`*`)`. For instance, to put `(2,3)` on the board, you call `put( (2,3) )`. Note the double parentheses!)

To retrieve *`my_tuple`* from the board, you call `get(`*`my_tuple`*`)`. In practice, though, you typically don't know yet what tuple you want to retrieve. For that reason, `get()` allows wildcards, in the form of `None` elements in its tuple argument. For instance, `get( (2,None) )` means: retrieve from the board any tuple with two elements of which the first is `2` and the second may be anything at all.

The `get()` function is blocking. This means that if there is no matching tuple on the board, it will sit around waiting until one appears. This is what happens in the example code: the first thread (the one with `me()==0`) sits around waiting for tuples of the form `("HELLO",` *`<something>`*`)` to show up on the board. Every time one such tuple shows up, the `get()` function returns it, the loop loops, and the `get()` function is called again, blocking until the next tuple shows up.

When a tuple is gotten with `get()`, it is removed from the board. Therefore, in the above code, you do not need to worry about looping around retrieving the same tuple over and over again. If you want to retrieve a tuple and leave it available (maybe for some other thread), you simply put it back on with `put()`.

Another important trick in the use of `get()` is that you can specify more than one pattern. For instance, `get( ("HELLO,"), ("ERROR",) )` blocks until either the one-element tuple `("HELLO",)` or the one-element tuple `("ERROR",)` shows up on the board, then removes and returns the tuple.

Finally, I will mention `grab()`. It is like `get()`, except that it does not block. If a matching tuple is available, it will return that, but if there is none, it will simply return `None` (unlike `get()`, which will sit there twiddling its thumbs until a matching tuple shows up).

> You could, conceivably, have your own shared data structures simply by declaring some global variables outside of `session()`, but I recommend that you don't. In order to make sure your data structures remain consistent, you would have to implement rules to make sure that no two threads fiddle with them at the same time ("mutual exclusion"), which can rapidly become extremely complicated. The Board is a data structure known as a Tuple Space, which already has mutual exclusion built into it.

10. Lesson 5: Wherein we deal with input

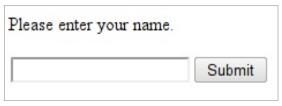
Now that we have seen how to manipulate what appears on a client screen, we will turn our attention to the issue of input.

There are two basic ways of giving input to a Willow client: you can click things, and you can fill out text fields.

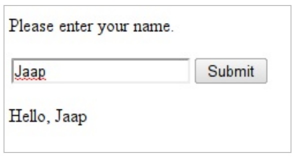
The `lesson05.py` code demonstrates both.

```
from willow.willow import *

def session():
  add("<p>Please enter your name.")
  add("<input id='bid' type='text'>")
  add("<input id='submit' type='submit'>")
  _, _, _ = get(("click", me(), "submit"))
  peek("#bid")
  _, _, name = get(("peek", me(), None))
  add("<p>Hello, %s." % name)

run(session)
```

This should initially display a screen that looks like so:



You can then type something in the text box and click the button labeled "Submit", and the screen will look like:


Please enter your name.

Jaap

Hello, Jaap

What's going on here?

First of all, HTML elements that look like

```
<input type='submit' ...>
```

are buttons.

In HTML, each `<input>` element that is a button must have two attributes: `type` must be set to `submit`, to make sure we are displaying a button and not some other sort of input element, and `value` must be set to the text that is to be on the button (or, if it is left out, the text defaults to "Submit", or possibly something else if your browser is set up for a language other than English).

In Willow, your button must additionally have an `id` attribute, which is how we keep track of which buttons get clicked.

When a button gets clicked, a tuple `("click", n, id)` is automatically posted onto the board, where *n* is the number of the client that received the button click and *id* is the `id` of the button.

This is why, after we have displayed the button, we can just call `get( ("click", me(), "submit") )`, which will block until the button is clicked. Note that if you insist on blocking one client thread until a button in some *other* client thread has been clicked, you can do that, simply by replacing `me()` in the above call to `get()` by some other integer (but it is not recommended).

If the button is clicked multiple times, multiple tuples will be inserted. This may result in behavior you may not necessarily intend. If you want to wait for a button to get clicked that may have been clicked before you started waiting, you will need to clear those old tuples off the board, by saying e.g. `grab( ("click",me(),None) )`.

The way we deal with text fields is similar to the way we deal with buttons, but there is no natural moment to automatically insert the contents of a text field into the Board, so instead we have to request it explicitly.

To show a text field, we use a piece of HTML that looks like

```
<input id='_name_of_text_field_' type='text'>
```

and then we call `peek("#name_of_text_field")`. This will cause a tuple to be posted to theboard of the form `("peek", n, text)`, where *n* is the number of the client and *text* is the text that the user typed in the field. In the above case, we only start looking for what's in the field after the user has pressed the button.

11. Lesson 6: Wherein we poke around in other threads

Many of the Willow functions, even though by default they operate on the client display associated with the current session thread, can also operate on a different client display if you pass the `number=`*n* optional parameter. This is never necessary, but it may be more convenient than passing messages through the board to the other session thread so that it can then manipulate its corresponding client display.

Take for instance the code in lesson 4:

```
from willow.willow import *

def session():
  if me() == 0:
    add("<h1>Monitor</h1>")
    while True:
      _, n = get(("HELLO", None))
      add("<p>Client %d logged in" % n)
  else:
    put(("HELLO", me()))
    add(me())

run(session)
```

We could rewrite that as follows:

```
from willow.willow import *

def session():
  if me() == 0:
    add("<h1>Monitor</h1>")
  else:
    add("<p>Client %d logged in" % me(), number=0)
    add(me())

run(session)
```

Strictly speaking, the above programs are not entirely equivalent. If the first session thread executed very, very slowly and the second client connect immediately after the first, then the "Client 1 logged in" message could conceivably show up before the "Monitor" heading when running the second program, but not when running the first.

12. Lesson 7: Wherein we wrap up some loose ends

We have now encountered virtually the whole set of Willow primitives. If you look at the API documentation, you will discover a few odds and ends that haven’t been mentioned so far.

The `hold()` and `flush()` functions

These may improve performance in very high-throughput applications but most likely aren't anything you need to worry about.

The optional `delay` argument to `put()`

You will also see that `put()` accepts a delay as an optional argument: calling `put( ("boo",), delay=3.4 )` causes the tuple `("boo",)` to be posted to the board 3.4 seconds later; the `put()` function itself returns immediately. This makes it possible, for instance, to do things like "wait for this button to be clicked or 7.2 seconds to elapse, whichever comes first," like so:

```
add("<input id='submit' type='submit'>")
put(("timeout",), delay=7.2)
get(("click", me(), "submit"), ("timeout",))
```

The `config()` function

You may want to use configuration files to configure, say, different treatments. In order to make this possible, Willow has a function that reads a JSON object out of a text file.

The `log()` function

This is how you cause your Willow program to produce a data file. Each call to `log()` will result in a single row in a CSV file, which is a simple spreadsheet format that is understood by Excel, OpenOffice, Stata, and other data analysis software. Each row will automatically get a timestamp added to the front of it, which is a large integer representing the number of seconds that have passed since the "Unix epoch" on January 1, 1970, 0:00 UTC. This may seem like an odd choice of timestamp format, but the advantage of this type of timestamp is that it makes sense to subtract them! The log file will be put in the `log/` subdirectory (folder) and it will be named automatically based on the time your Willow program has started. (A more human-readable timestamp format is used for the filenames.)

13. API: willow.willow

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.

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.

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

13.1. `me()`

Returns the number of the current client.

13.2. `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`".

13.3. `set(content, [selector], [number])`

Set the content of the HTML element(s) referred to by `selector` on to be `content`.

13.4. `add(content, [selector], [number])`

Add `content` to the content of the HTML element(s) referred to by `selector`.

13.5. `tweak(value, attribute, [selector], [number])`

Change the value of attribute `attribute` to be `value` in HTML element(s) referred to by `selector`.

13.6. `push(style, [selector], [number])`

Add the CSS style `style` to the HTML element(s) referred to by `selector`.

13.7. `pop(style, [selector], [number])`

Remove the CSS style `style` from HTML element(s) referred to by `selector`.

13.8. `hide([selector], [number])`

Hide the HTML element(s) referred to by `selector`.

13.9. `show([selector], [number])`

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

13.10. `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.

13.11. `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.

13.12. `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.

13.13. `put(tuple, [delay])`

After an optional `delay` in seconds, post the tuple onto the communication board.

> 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,)`.

13.14. `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.

13.15. `grab(*queries)`

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

13.16. `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.

13.17. `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}`.

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 about that. To get the value of the `rounds` parameter, simply use `cfg["rounds"]`.

13.18. `log(x,y,…)`

Write (x,y,…) to the log file as a record, with a UNIX timestamp prepended as the first column. 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. For instance, if Willow was invoked at 15:32:42 on February 2 of the year 2010, the file would be called `log/2010-02-10-15-32-42.csv`. This particular date-and-time format is used so that the files can be sorted by date/time. The log files are in CSV format and can be opened with a text editor or a spreadsheet program.

13.19. CSS features

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.

14. API: willow.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` values that start with `__` (two underscores).

If you plan to use Twig, you should avoid using any tuples or HTML `id` values 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.)

14.1. `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)
```

You can also pass an integer argument `n` to `assemble()`. This indicates that you want exactly n clients besides the monitor (n+1 total), and thus allows you to skip the whole button-clicking step.

14.2. `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)
```

15. Appendix: 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.

15.1. 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)
$
```

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.

15.2. How to find your IP address on Windows

This section has yet to be written.

16. Appendix: 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.

17. Appendix: Architecture

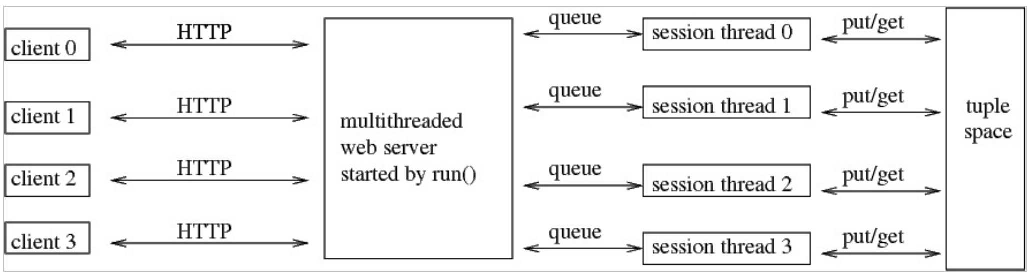
In this section, I will describe some of the nuts and bolts of Willow. It is not usually necessary to understand this, but it may be helpful for those seeking to extend Willow or satisfy their curiosity.

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 updated 2010-02-12 16:20:28 EDT