Ibex 0.3.6 Manual

Alex Drummond

Contents

Check out the printable PDF version of this manual.

Versions

This documentation covers 0.3.7. The 0.1.x versions of Webspr come with a README containing full documentation. 0.2.x versions are documented in another page on the wiki. The latest version of this document can be found at https://github.com/addrummond/ibex/blob/master/docs/manual.md

See the end of this document for the changelog.

Prior to version 0.3 this software was called "webspr", not "Ibex".

Requirements

- Python >= 2.3. (Python 3000 not currently supported.)

Introduction

Ibex ("Internet Based EXperiments") allows you to run various kinds of psycholinguistic experiment online. The system is modular and the range of possible experiments is steadily growing. Ibex uses JavaScript and HTML only, and doesn't require the use of plugins such as Java or Flash.

Setting up the server

Note: If you are using the Ibex Farm, you can skip to the "Basic Concepts" section below. For most applications you will only need to modify the file `example_data.js`. (The Ibex Farm allows you to modify/upload some other files, but this rarely necessary.)

There are two ways of running the server: either using the stand-alone toy HTTP server, or as a CGI process. By default, the server runs in stand-alone mode. To start it, change to the `www` directory and execute server.py:

```
python server.py
```

This will start the server on port 3000 by default (so the experiment will be at http://localhost:3000/experiment.html).

The stand-alone server is limited to serving files relating to Ibex, so it is unlikely to be useful for real work (unless perhaps you hide it behind a proxy). However, it is useful for testing experimental designs without setting up a real HTTP server, and for running experiments offline.

The server can be configured by editing `server_conf.py`. Each of the options is commented. Change the `SERVER_MODE` variable to determine whether the server runs in stand-alone mode or as a CGI application. When `SERVER_MODE` is set to "cgi", `server.py` will work as a standard CGI program; when it is set to "toy", the server will run in stand-alone mode. (The value "paste" has the same effect as "toy", for backwards compatibility with 0.2.x version.) The value of the `PORT` variable determines the port the server will listen on when operating in stand-alone mode. As an alternative to editing the configuration file, the `-m` and `-p` command-line options can also be used to set the server mode and port respectively. Command-line options override those set in `server_conf.py`.

New: Version 0.3 adds some new ways of configuring the server. These are unlikely to be of interest to most users, but are documented in the "New Configuration Methods" section below.

If running the server as a CGI application, make sure that the web server can serve up `experiment.html`, and the other files in the `www` dir, as static files. All dynamic requests go through `server.py`, so as long as the files in `www` can be accessed, and the server recognizes `server.py` as a CGI application, everything should work. The "Step-by-step CGI" section below goes through the process of setting up webspr as a CGI app.

The directories `js_includes`, `data_includes` and `css_includes` contain JavaScript and CSS files required for particular kinds of experiment. When an HTTP request is made for `server.py?include=js`, `server.py?include=data` or `server.py?include=css`, the server concatenates all the files in the specified directory and serves up the result. Your data file(s) will live in `data_includes`. **Note:** The server mangles CSS files by adding a prefix to each class/id name based on the name of the CSS file; see the section "CSS mangling" below for more details.

The server stores some state in the `server_state` directory. This directory is automatically created by the server if it does not already exist, and in order to start the server from a fresh state you may simply delete this directory and/or its contents. Currently, the directory just contains the counter used to alternate latin square designs (see "Latin Square Designs" section below). This counter can be reset on server startup using the `-r` option.

The server logs messages in the file `server.log`. The default is to store all files in the same directory as `server.py` (i.e. logs, results, the `server_state` directory, etc.), but the working directory of the server can be modified by setting the variable `IBEX_WORKING_DIR` in `server_conf.py`. You can also set the environment variable of the same name. The value of this environment variable overrides the value of the variable in `server_conf.py`.

On Linux/Unix/OS X, the server uses file locking to ensure that the results file and server state remain consistent. Currently, it does not do so on Windows, so there is a theoretical possibility of the server state or the results being corrupted if the server is deployed in a Windows environment (but unless there is very high traffic, it's purely theoretical).

The stand-alone server has been tested on Windows and OS X (but will almost certainly work on any system with Python 2.3-2.6). The CGI server has been tested on OS X and Linux using the lighttpd web server in both cases.

New Configuration Methods

Version 0.3 of this software introduces two new methods of setting configuration variables. The first is simply to include the variable definitions in `server_conf.py` directly in `server.py`, obviating the need to set the `SERVER_CONF_PY_FILE` variables in `server.py`. The second is to specify that `server.py` should issue an HTTP GET request to a given URL, with the result obtained being interpreted as a JSON dictionary specifying the values of the configuration variables. This mode is configured by setting the configuration variables `EXTERNAL_CONFIG_URL`, `EXTERNAL_CONFIG_METHOD` and `EXTERNAL_CONFIG_PASS_PARAMS`. The first of these variables is self-explanatory; the second must be set to `"GET"`; the third should be set to a boolean, specifying whether or not `server.py` should pass a `dir` paramater in the query string giving the directory in which `server.py` resides. Note that the values of these three variables may be set either in `server_conf.py`, or directly in `server.py`.

The new configuration methods are used to handle the configuration of experiments in the Ibex Farm (http://spellout.net/ibexfarm). They are unlikely to be of interest to other users.

Step-by-step CGI

This subsection describes how to set up webspr as a CGI application from scratch. Since the particulars of different web servers differ, some parts are necessarily rather vague. There are many possible ways of organizing the files/directories; the setup described below is just the simplest.

1. Ensure that the `www` directory is somewhere within the root directory of your HTTP server. You may want to place the entire `webspr` directory within the root directory, or you may want to copy the files in `www` to a new location within the root directory. All of these files are ordinary static files except for `server.py`, which needs to be run as a CGI app.
2. Decide on a location for the main webspr directory. Depending on the details of your setup, you may be able to leave it in place, or you may need to copy it to a directory which the HTTP server has permission to access (so that `server.py` may access files in this directory when it is executed).
3. Edit `server.py` and change the value of `SERVER_CONF_PY_FILE` to point to the new location of server_conf.py.
4. Edit `server_conf.py` and change the value of `IBEX_WORKING_DIR` to the new location of the main `webspr` dir.
5. Edit `server_conf.py` and change change the value of `SERVER_MODE` to `"cgi"`.
6. If the version of Python you wish to use is not the default used by the HTTP server, add a `#!` line at the beginning of `server.py`.
7. You may need to rename `server.py` (e.g. some servers may only execute CGI scripts with the extension `.cgi`). If this is the case, follow the instructions in the subsection below.

Once these steps are complete, it's just a matter of configuring your HTTP server correctly. If you are using shared hosting, there is a reasonably good chance that the HTTP server will already have been configured to run Python CGI apps, in which case you won't need to do any additional configuration. There is an example configuration file for lighttpd included in the distribution (`example_lighttpd.conf`).

Renaming server.py

Ibex will stop working if you rename `server.py`, since the files `experiment.html` and `overview.html` assume that the script has this name. (In older versions of ibex there were a few other files which made this assumption, but as of 0.3-beta14, this has been changed.)

You can edit `experiment.html` and `overview.html` manually to replace all instances of `server.py` with the new name of the script. Alternatively, you can execute the renamed `server.py` script with the following options to automatically generate new `experiment.html` and `overview.html` files:

```
python /path/to/renamed/server.py --genhtml /foo/webspr-XXXX/www
```

This command will produce no output if it executes successfully. Note that you must run this command **after** renaming `server.py`.

Basic concepts

An Ibex experiment is essentially a sequence of items. Each item is itself a sequence of entities. To give a concrete example, each item in a self-paced reading experiment might consist of two entities: an entity providing a sentence for the subject to step through word by word, followed by an entity posing a comprehension question. Some other items and entities would also be present under most circumstances. For example, there would probably be a "separator" entity between each sentence/question pair in order to provide a pause between sentences. Schematically, the sequence of items would be as follows (the Wiki insists on displaying this in pretty colors):

```
ITEM 1:
    ENTITY 1: Sentence
    ENTITY 2: Comprehension question
ITEM 2:
    ENTITY 1: Pause for two seconds
ITEM 3:
    ENTITY 1: Sentence
    ENTITY 2: Comprehension question
ITEM 4:
    ENTITY 1: Pause for two seconds
ITEM 5:
    ENTITY 1: Sentence
    ENTITY 2: Comprehension question
```

It is not necessary to construct the full sequence of items and entities manually. Ibex provides tools for ordering items in various ways (e.g. for inserting a pause between each item) and for implementing latin square designs. More on these shortly.

Each entity is an instance of a controller, which determines the kind of experimental task that the entity will present. For example, there is a `DashedSentence` controller useful for self-paced reading and speeded acceptability judgment tasks. Ibex has a modular design, where each controller is a JavaScript object that follows a standard interface. This makes it quite easy to add new controllers if you are familiar with JavaScript/DHTML programming.

Ibex stores results in CSV format. This makes it easy to import results into spreadsheets, Matlab, R, etc. Each entity may contribute zero, one **or more** lines to the results file. The first seven columns of each line give generic information about the result (e.g. an MD5 hash identifying the subject) and the rest give information specific to the particular element (e.g. word reading times, comprehension question answers). The first seven columns are as follows:

Column	Information
1	Time results were received (seconds since Jan 1 1970)
2	MD5 hash identifying subject. This is based on the subject's IP address and various properties of their browser. Together with the value of the first column, this value should uniquely identify each subject.
3	Name of the controller for the entity (e.g. "DashedSentence")
4	Item number
5	Element number
6	Type
7	Group

Note that some information pertaining to an entire experiment (e.g. the time that the server received the results) is duplicated on each line of the results. This often makes it easier to parse the results, though it makes the format much more verbose. The significance of the "type" and "group" columns will be explained later; they are involved in specifying the ordering of items.

There are two ways to find out what the values in each column mean. The "manual" method is to interpret the first seven columns of each line as specified above, and then to consult the documentation for the relevant controllers to determine the format of the subsequent columns. All of the controllers that are bundled with the Ibex distribution are fully documented here. An easier method is to look at the comments in the results file: for each sequence of lines in the same format, the server adds comments describing the values contained in each column.

Important: A single element my contribute multiple lines to the results file. For example, if the `DashedSentence` element is in "self-paced reading" mode, it will add a line to the results file for every word reading time it records. Thus, there is not a one-to-one correspondence between items/elements and lines.

New: The server now uses a more sophisticated algorithm for commenting lines in the results file. It is now able to handle cases where there are repeating patterns of lines in a results file.

Newer: As of 0.3-beta15, there is another commenting algorithm available, which can be turned on by setting the `SIMPLE_RESULTS_FILE_COMMENTS` variable to `True` in `server_conf.py` (it defaults to `False`). This algorithm simply adds a comment for every line in the results file. This simpler method generally works better than the old algorithm, which tried to be clever but didn't really succeed. If you are using the Ibex Farm, there is currently no way of changing this configuration option using the web interface, but this should be fixed soon.

Format of a data file

Data files for Ibex are JavaScript source files, but it is not really necessary to know JavaScript in order to write one. The most important part of a data file is the declaration of the "items" array, as in the following example:

```
var items = [

["filler", "DashedSentence", {s: "Here's a silly filler sentence"}],
["filler", "DashedSentence", {s: "And another silly filler sentence"}],
["relclause", "DashedSentence", {s: "A sentence that has a relative clause"}],
["relclause", "DashedSentence", {s: "Another sentence that has a relative clause"}]

]; // NOTE SEMICOLON HERE
```

The "items" array is an array of arrays, where each subarray specifies a single item. In the example above, every item contains a single element (multiple element items will be covered shortly).

- The first member of each subarray specifies the type of the item. Types can either be numbers or strings. Although the types in the example above have descriptive names, Ibex does not interpret these names in any way.
- The second member specifies the controller.
- The third member is an associative array of key/value pairs. This array is passed to the controller and is used to customize its behavior. In this case, we pass only one option ("s"), which tells the `DashedSentence` controller which sentence it should display.

Once the items array has been created, Ibex must be told the order in which the items should be displayed. There are some moderately sophisticated facilities for creating random orderings and latin square designs, but for the moment, let's just display the items in the order we gave them in the array. This can be achieved by adding the following definition (which won't make much sense yet):

```
var shuffleSequence = seq(anyType);
```

Suppose we wanted to pair each sentence with a comprehension question. The easiest way to do this is to add a second element to each item:

```
var items = [

["filler", "DashedSentence", {s: "Here's a silly filler sentence"},
           "Question", {q: "Is this a filler sentence?", as: ["Yes", "No"]}],
["filler", "DashedSentence", {s: "And another silly filler sentence"},
           "Question", {q: "Does this sentence have a relative clause?", as: ["Yes","No"]],
["relclause", "DashedSentence", {s: "A sentence that has a relative clause"},
              "Question", {q: "Was there movement through [Spec,CP]?", as: ["Yes", "No"]}],
["relclause", "DashedSentence", {s: "Another sentence that has a relative clause"},
              "Question", {q: "Was the first word 'frog'?", ["Yes", "No"]}]

]; // NOTE SEMICOLON HERE
```

As shown above, this is done simply by adding to each item further pairs of controllers and associative arrays of options. It is rather tiresome to have to set the `as` option to ["Yes", "No"] every time, but this can easily be avoided by specifying this as the default for the Question controller. To give an example of how defaults for multiple controllers are specified, let's also set the `mode` option of `DashedSentence` to "speeded acceptability":

```
var defaults = [
    "Question", {as: ["Yes", "No"]},
    "DashedSentence", {mode: "speeded acceptability"}
];
```

Once this definition is in place, we can drop the specification of the `as` option for each Question element. The final data file looks like this:

```
var shuffleSequence = seq(anyType);

var defaults = [
    "Question", {as: ["Yes", "No"]},
    "DashedSentence", {mode: "speeded acceptability"}
];

var items = [

["filler", "DashedSentence", {s: "Here's a silly filler sentence"},
           "Question", {q: "Is this a filler sentence?"}],
["filler", "DashedSentence", {s: "And another silly filler sentence"},
           "Question", {q: "Does this sentence have a relative clause?"}],
["relclause", "DashedSentence", {s: "A sentence that has a relative clause"},
              "Question", {q: "Was there movement through [Spec,CP]?"}],
["relclause", "DashedSentence", {s: "Another sentence that has a relative clause"},
              "Question", {q: "Was the first word of that sentence 'frog'?"}]

];
```

Where to put your data file: Data files live in the `data_includes` directory, and must have a `.js` extension. You can only have data one file in the directory at any one time, since if you have multiple files, they will just overwrite the definitions for `shuffleSequence`, `items` and `defaults`. However, you can tell the server to ignore some of the files in `data_includes` (see the section "Configuring `js_includes`, `data_includes` and `css_includes`").

Shuffle sequences

`seq`, `randomize` and `shuffle`

A "shuffle sequence" is a JavaScript data structure describing a series of "shuffling", randomizing and sequencing operations over an array of items. A variable called `shuffleSequence` should be defined in your data file with a data structure of this sort as its value.

Shuffle sequences are composed of three basic operations, `seq`, `randomize` and `shuffle`. Both take a series of "type predicates" as arguments, where each type predicate is the characteristic function of a set of types. A type predicate may be one of the following:

- A string or integer, denoting the characteristic function of all items of the given type.
- A JavaScript function, returning a boolean value when given a type (either a string or integer).
- Another shuffle sequence (shuffle sequences can be embedded inside bigger shuffle sequences).

The basic operations work as follows:

- A statement of the form `seq(pred1, pred2, pred3, ...)` specifies a sequence where all items matching `pred1` precede all items matching `pred2`, all items matching `pred2` precede all items matching `pred3`, and so on. The original relative ordering between items of the same type is preserved. A `seq` with only one argument is permissible.
- A statement of the form `randomize(pred1)` specifies a randomly ordered sequence of all items matching pred1.
- A statement of the form `shuffle(pred1, pred2, pred3, ...)` specifies that items matching the given predicates should be shuffled together in such a way that items matching each predicate are evenly spaced. The original relative ordering between items of the same type is preserved.

The following type predicates are predefined as JavaScript functions:

Function	Description
`anyType`	Matches any type.
`lessThan0`	Matches any integer type < 0.
`greaterThan0`	Matches any integer type > 0.
`equalTo0`	Matches any integer type = 0.
`startsWith(s)`	Matches any string type starting with s.
`endsWith(s)`	Matches any string type ending with s.
`not(pred)`	Matches anything that is not of a type matched by pred.
`anyOf(p1, p2, ...)`	Takes any number of type predicates as its arguments, and matches anything matching one of these predicates.

If you define your own predicates, be careful to test that they are cross-browser compatible. See the "Cross-browser compatibility" section below for some pertinent advice. The predicates above are defined in `shuffle.js`.

The power of shuffle sequences derives from the possibility of composing them without limit. Suppose we want the following order: all practice items in their original order followed by evenly spaced real and filler items in random order. Assuming the types "practice", "real" and "filler", we could use the following shuffle sequence:

```
seq("practice", shuffle(randomize("real"), randomize("filler")))
```

Now suppose that there are two types of real item ("real1" and "real2"), and we wish to order the items the same way as before:

```
seq("practice", shuffle(randomize(anyOf("real1", "real2")),
                randomize("filler")))
```

What if we also want the two types of real item to be evenly spaced? The following formula will do the trick:

```
seq("practice", shuffle(randomize("filler"),
                        shuffle(randomize("real1"),
                                randomize("real2"))))
```

This first shuffles items of type "real1" and items of type "real2" and then shuffles filler items into the mix. Finally, practice items are prepended in the order they were given in `data.js`.

Since it is often useful to apply `randomize` to every argument of `shuffle`, there is a utility function, `rshuffle`, which automates this. The following equivalence holds:

```
rshuffle(a1, a2, ...) = shuffle(randomize(a1), randomize(a2), ...)
```

If no shuffle sequence is specified, Ibex uses the following default sequence:

```
seq(equalTo0, rshuffle(greaterThan0, lessThan0))
```

This will seem rather cryptic to anyone not familiar with the behavior of earlier versions of Ibex, where this ordering specification was built in and unchangeable. In short, it works well if practice items have type 0, filler items have integer types < 0, and real items have integer types > 0.

Important: A shuffle sequence must always have one of `seq`, `shuffle`, `randomize` or `rshuffle` as its outer element. A single string or integer is not a valid shuffle sequence, and neither is a predicate expression such as `not("foo")`. Thus, one must use `seq("foo")`, not just `"foo"`, and `seq(not("foo"))`, not just `not("foo")`.

It is possible to include duplicate items in the final sequence. For example, `seq("foo", "foo")` would include every item of type "foo" twice. For this reason, it is possible to accidentally include duplicate items if a number of your predicates overlap. You can define your own shuffle sequence operators and predicates quite easily; see `shuffle.js` for the definitions of the `seq`, `shuffle` and `randomize` operators.

Adding separators

Normally, it is a good idea to have some sort of padding in between items to warn participants that they have finished one item and are starting another. Ibex provides the `Separator` item for this purpose. It can either work on a timeout or prompt for a keypress. Here's an example:

```
["sep", "Separator", {transfer: 1000, normalMessage: "Please wait for the next sentence."}]
```

When the `transfer` option is set to 1000, this specifies that there should be a 1000ms wait before the next item. The `normalMessage` option gives the message that should be displayed if the participant didn't do anything wrong on the previous item (see the section "Communication between elements" for more details). If the other items have failure conditions, you should set the `errorMessage` option too. If you want the participant to proceed by pressing a key rather than waiting, set `transfer` to "keypress".

The shuffle sequence operator `sepWith` is provided for the purpose of interpolating separators with other items. It takes two arguments: the first is a shuffle sequence specifying the sequence of items which should be used to separate the other items; the second argument is a shuffle sequence specifying the other items.

Let's take the example data file above and add timeout Separators between each item. There are no failure conditions in this experiment (since there is no "wrong" answer for an acceptability judgment) so we only need to specify the `normalMessage` option. For the moment, we'll still present the sentences in the order in which they appear in the `items` array. Here's the modified file:

```
var shuffleSequence = sepWith("sep", not("sep"));

var defaults = [
    "Question", {as: ["Yes", "No"]},
    "DashedSentence", {mode: "speeded acceptability"}
];

var items = [

["sep", "Separator", {transfer: 1500, normalMessage: "Please wait for the next item."}],

["filler", "DashedSentence", {s: "Here's a silly filler sentence"},
           "Question", {q: "Is this a filler sentence?"}],
["filler", "DashedSentence", {s: "And another silly filler sentence"},
           "Question", {q: "Does this sentence have a relative clause?"}],
["relclause", "DashedSentence", {s: "A sentence that has a relative clause"},
              "Question", {q: "Was there movement through [Spec,CP]?"}],
["relclause", "DashedSentence", {s: "Another sentence that has a relative clause"},
              "Question", {q: "Was the first word of that sentence 'frog'?"}]

];
```

Manipulating individual items

So far, we've been treating items as atoms for the purposes of shuffle sequences – although an item might be composed of several elements, the operations `seq`, `randomize`, `shuffle` and `sepWith` ignore this internal structure. However, it is sometimes useful to be able to append or prepend a particular sequence of elements to every item. For example, if you are doing a speeded acceptability judgment task, you might want every sentence to be followed by exactly the same question ("Was this a good sentence?"). To this end, webspr provides the `precedeEachWith` and `followEachWith` operations.

Both functions take two arguments, each of which should be a shuffle sequence. The first argument specifies the sequence of items which (when flattened to a sequence of elements) is to be appended/prepended to every item in the second argument. Returning to the previous example data file, let's modify it so that every sentence is followed by the same acceptability question. We'll also ensure that the sentences are randomly ordered, with the fillers and "real" sentences evenly spaced:

```
var shuffleSequence = followEachWith("question", rshuffle("filler", "relclause"));

var defaults = [
    "DashedSentence", {mode: "speeded acceptability"}
];

var items = [
```

```
["question", "Question", {q: "Was that a good sentence?", as: ["Yes", "No"]}],

["filler", "DashedSentence", {s: "Here's a silly filler sentence"}],
["filler", "DashedSentence", {s: "And another silly filler sentence"}],

["relclause", "DashedSentence", {s: "A sentence that has a relative clause"}],
["relclause", "DashedSentence", {s: "Another sentence that has a relative clause"}]

];
```

Important: `precedeEachWith` and `followEachWith` have a somewhat confusing behavior with respect to the format of the results file. Although in effect, the first argument of `followEachWith` is appended to every item, the appended elements are not considered a part of the items they are appended to in the results file. Rather, every question in the preceding example will have the same item number and element number in the results file (that based on the position of the question item in the data file).

Latin square designs

Ibex has built-in support for latin square designs. These are implemented by assigning each item a group in addition to its type. Each participant sees only one item out of all the items in any given group. To give an example, we could place both of the "relclause" sentences from the previous example in the same group using the following code:

```
[["relclause", 1], "DashedSentence", {s: "A sentence that has a relative clause"}],
[["relclause", 1], "DashedSentence", {s: "Another sentence that has a relative clause"}]
```

Now, any given participant will see only one of these sentences. Designs are rotated using a counter stored on the server. Groups, like types, may either be strings or numbers. By default, webspr does not check that all groups contain the same number of items (this behavior is new in 0.2.4). You can set the `equalGroupSizes` variable to `true` in order to revert to the old behavior, where an error is raised if groups contain differing number of items. (See the "Miscellaneous options" subsection below.)

New: You may now choose which latin square a participant will be placed in by using a URL of the following form: ". . . /server.py?withsquare=XXXX". This will set the latin square to XXXX for the current participant and then display the main experiment page. Selecting the latin square in this manner does **not** modify the master counter stored on the server.

Newer: You can also set the counter in the data file, with a statement such as `var counterOverride = 18;`. Again, the server's latin square counter is not modified. This method takes priority over the URL method described in the previous paragraph, if both are used.

A new (and not very well-tested) feature allows for more complex designs where selection of an item from one group is dependent on selection of an item from another. For example, you can specify that the item chosen from group 3 should be the same as the item chosen from group 2 (i.e. if the first item is chosen from group 2, the first item will also be chosen from group 3). To use this feature, just replace each group specifier with a pair of group specifiers, where the first member of the pair is the original group specifier, and the second is the group which governs selection from the original group. For example, suppose that we have another group (group 2), and we want choices from this group to be linked to choices from group 1 in the example above. The following code will do the trick:

```
[["relclause", [2, 1]], "DashedSentence",
  {s: "I am paired with 'A sentence that has a relative clause'"}],
[["relclause", [2, 1]], "DashedSentence",
  {s: "I am paired with 'Another sentence that has a relative clause'"}]
```

Note that the second number in the pair must be the same for all items in the group. **Warning:** This feature may be removed in future releases if I get around to implementing a more general way of implementing more complex latin square designs.

Sending results early

You can now control the point in the experiment at which Ibex sends the results to the server. This allows you to (e.g.) present the participant with a link to another website after the results have been successfully uploaded. In order to use this feature, you must first set the `manualSendResults` config variable to `true`. Then, add a `__SendResults__` controller to your items list and insert it somewhere in your shuffle sequence. There is some example code in `data_includes/example_data.js`. Note that you may only send results once per experiment – it is not possible to incrementally upload results.

If you set `manualSendResults` to `true`, but do not add a `__SendResults__` controller to your shuffle sequence, then the results will never get sent at all.

Modifying the running order manually

Occasionally, shuffle sequences aren't powerful enough to arrange items in the order you wish. To perform abritrary rearrangements of items, you can define a `modifyRunningOrder` function in your data file. This function takes as its input the running order generated by the shuffleSequence, and returns a new runningOrder. (The function is permitted to modify its argument, but it must return a running order.) A running order is an array of arrays of elements. Each element is an object with the following properties:

- `itemNumber`
- `elementNumber`
- `type`
- `group`
- `controller` (a string giving the name of the controller)
- `options` (an object giving the options for the controller)
- `hideResults` (boolean; if true, results from this controller are not included in the results file)

You can create new objects with these properties and insert them into the running order. Typically, these objects would have `itemNumber`, `elementNumber`, `type` and `group` set to `null`. In this case, if the controller adds lines to the results file, then the "item", "element", "type" and "group" columns will all have the value "DYNAMIC".

The convenince constructor `DynamicElement` is provided to ease the construction of typical element objects. This takes as its first argument a controller name, and as its second the controller's options. Optionally, `true` may be passed as the third argument to set the `hideResults` property to `true`. The following example inserts a "pause" `Message` at every tenth item:

```
function modifyRunningOrder(ro) {
    for (var i = 0; i < ro.length; ++i) {
        if (i % 10 == 0) {
            // Passing 'true' as the third argument casues the results from this controller
            // to be omitted from the results file. (Though in fact, the Message controller
            // does not add any results in any case.)
            ro[i].push(new DynamicElement(
                "Message",
                { html: "<p>Pause</p>", transfer: 1000 },
                true
            ));
        }
    }
    return ro;
}
```

Communication between elements

Ibex allows for a limited amount of communication between elements. Each element may set keys in a hashtable which is passed to the next element. The hashtable is cleared between elements, so there is no possibility of long-distance communication.

Currently, this system is used to provide feedback to participants when they (for example) answer a comprehension question incorrectly. Controllers such as `Question` set the key "failed" if something goes wrong, and the next `Separator` item is then able to display a message warning the participant that they answered incorrectly.

Controllers

For all controllers, the `hideProgressBar` option may be set to true, in order to prevent the progress bar being displayed while the controller is in action. Similarly, the `countsForProgressBar` option may be set to determine whether or not completion of the controller causes the indicator in the progress bar to move to the right. (Different controllers have a different default setting for `countsForProgressBar`.)

Separator

Options

Option	Default	Description
transfer	`"keypress"`	Must be either `"keypress"` or a number. If the former, participant goes to the next item by pressing any key. If the latter, specifies number of ms to wait before the next item.
normalMessage	`"Press any key to continue."`	Message to display if the previous item was completed normally.
errorMessage	`"Wrong. Press any key to continue."`	Message to display (in red) if the previous item was not completed normally (e.g. timeout, incorrect answer).
ignoreFailure	`false`	If true, never displays an error message.

Results

`Separator` does not add any lines to the results file.

Message

Options

Option	Default	Description
html	obligatory	The HTML for the message to display (see section "HTML Code" below).

Option	Default	Description
transfer	`"click"`	Either `"click"`, `"keypress"`, an integer, or `null`. If `"click"`, the participant clicks a link at the bottom of the message to continue (see"continueMessage" option). If `"keypress"`, they press any key to continue. If an integer, the experiment continues after pausing for the specified number of milliseconds. If `null`, there is no way for the user to complete the controller (and no "click to continue" message is displayed). This is useful only when the controller is part of a larger `VBox`.
consentRequired	`false`	If true, the participant is required to tick a checkbox indicating that they consent to do the experiment (in this case, the message would probably be some sort of statement of terms/conditions). This option can only be set to `true` if the "transfer" option is set go `"click"`. Note that it is also possible to create checkboxes like this using the more flexible `Form` controller.
continueMessage	`"..."`	Only valid if the "consentRequired" option is set to `"true"`. This specifies the text that will appear in the link that the participant needs to click to continue with the experiment.
consentMessage	`"..."`	Only valid if the "consentRequired" option is set to `"true"`. This specifies the text that will appear next to the checkbox.
consentErrorMessage	`"..."`	Only valid if the "consentRequired" option is set to `"true"`. This specifies the error message that will be given if the participant attempts to continue without checking the consent checkbox.

Results

`Message` does not add any lines to the results file.

DashedSentence

Options

Option	Default	Description
s	obligatory	The sentence. This is either a string, in which case the sentence will be presented word-by-word, or a list of strings ("chunks"), in which case the sentence will be presented chunk-by-chunk. If one of the words/chunks begins with the character `"@"`, then the controller will finish after the word beginning with "@" is displayed (the "@" will be stripped when the word is presented). This feature can be useful if you want (for example) to interrupt a self-paced reading item with a lexical decision task.
mode	`"self-paced reading"`	Either `"self-paced reading"` or `"speeded acceptability"`.
display	`"dashed"`	If set to `"dashed"`, the sentence is displayed as a sequence of dashes; when the participant presses the space bar, the current word is displayed above the corresponding dash, and no other words are displayed. If set to `"in place"`, words are displayed one-by-one in the center of the screen. Setting to `"in place"` changes the default value of `wordPauseTime` from 100 to 0 and the default value of `wordTime` from 300 to 400, since otherwise the display flickers.

Option	Default	Description
blankText	`"\u2014\u2014"`	Applicable only if "display" is set to `"in place"`. This is the text that is shown before the participant presses space for the first time. By default it is two horizontal dashes.
wordTime	`300`	If mode is `"speeded acceptability"`, the time in ms each word should be displayed for. (See also description of "display" option.)
wordPauseTime	`100`	If mode is `"speeded acceptability"`, the time in ms a word should remain blank before it is shown. (See also description of "display" option.)
sentenceDescType	`"literal"`	Determines the format of column 1 of the results (see table below). If `"literal"`, column contains the sentence itself (encoded as a URL with %XX escapes). Currently, this is the only possible value for this option.
showAhead	`true`	Whether or not to use dashes to indicate the presence of words ahead of the reader's current position.
showBehind	`true`	As above (but behind rather than ahead). If both showAhead and showBehind are set to `false`, no dashes are shown.
hideUnderscores	`false`	If this is set to `true`, the underscore character is interpreted as a space which does not trigger a break between words. This is an alternative to passing an array of strings as the value of the `s` option.

Note on right-to-left languages: The current version of Ibex doesn't provide out-of-the-box support for doing self-paced reading experiments with languages that are displayed right to left (although I hope to add support for this soon). However, it is possible to modify the code for the `DashedSentence` controller to display in right-to-left order. Please contact me (a.d.drummond@gmail.com) if you want to do this.

Results

The format of the results depends on the setting of the `mode` option. If it is set to "speeded acceptability", results have the following format:

Column	Description
1	See documentation for 'sentenceDescType' option above.

If `mode` is set to "self-paced reading", the results look like this:

Column	Description
1	Word number. For example, "1" indicates that this line gives the time it took to read the first word (i.e. the difference between and).
2	The word.
3	The reading time in ms.
4	Either 1 or 0. Indicates whether or not there was a line break between word n and word n + 1 (where reading time is for word n).
5	See documentation for 'sentenceDescType' option above.

FlashSentence

Options

Option	Default	Description
s	obligatory	The sentence to be displayed.
timeout	2000	If `null`, the sentence is displayed indefinitely (only useful if part of a VBox). Otherwise, a number giving the time in ms to display the sentence.
sentenceDescType	`"md5"`	See documentation for DashedSentence controller.

Results

Column	Description
1	See documentation for 'sentenceDescType' option of the DashedSentence controller.

Question

Options

Option	Default	Description
q	`null`	The question to pose.
as	obligatory	A list of strings giving the answers the user has to choose from. May also be a list of pairs of strings (e.g. `["y", "Yes"]`. In this case, the second string gives the answer, and the first string specifies a key which the user may press to select that answer. Note that these key assignments will be ignored if `presentAsScale` is `true` (but they will not be ignored if `presentHorizontally` is true).
instructions	`null`	Instructions for answering the question, displayed following the question and the answer selection.
hasCorrect	`false`	If `false`, indicates that none of the answers is privileged as correct. Otherwise, is either `true`, indicating that the first answer in the `as` list is the correct one; an integer, giving the index of the correct answer in the `as` list (starting from 0); or a string, giving the correct answer.
autoFirstChar	`false`	If set to true, users can select answers by pressing the key corresponding to the first letter in the answer. (This is useful e.g. for yes/no questions.)
showNumbers	`true`	If `true`, answers are numbered and participants can use number keys to select them.
randomOrder	`true` if the question has a correct answer, false otherwise.	Whether or not to randomize the order of the answers before displaying them. If `presentAsScale` or `presentHorizontally` are set to `true`, `randomOrder` can also be set to a list of keys. This list should be the same length as `as`. It specifies keys in a left-to-right order. This makes it possible for the mapping from keys to answers to correspond to the (random) order in which the answers are presented.
presentAsScale	`false`	If `true`, answers are presented as a scale (useful for acceptability ratings). If any of the points in the scale are integers 0-9, the participant may select them by pressing a number key.

Option	Default	Description
presentHorizontally	`false`	Note that at most one of `presentAsScale` and `presentHorizontally` should be set to `true`. `presentHorizontally` is like `presentAsScale` except that it does not override the usual means of specifying which key presses trigger which answers, and it leaves `randomOrder` on by default for questions which have correct answers. If your scale contains words, you may use the `autoFirstChar` option, and in all cases, you may explicitly specify answer keys, as described in the comment on the `as` option.
leftComment	`null`	Only valid if `presentAsScale` is `true`. This option specifies text to be displayed at the left edge of a scale (e.g. "Bad").
rightComment	`null`	As for "leftComment", but for the right edge of the scale.
timeout	`null`	If `null`, there is no time limit. Otherwise, should be a number giving the time in ms a participant has to answer the question.

Results

Column	Description
1	The question that was posed (encoded as a URL with %XX escapes).
2	The answer that the participant gave (also encoded as a URL).
3	"NULL" if the question as no correct answer. Otherwise, 1 if the participant answered correctly, or 0 if they didn't.
4	The time the participant took to answer the question (ms).

AcceptabilityJudgment

The `AcceptabilityJudgment` controller makes it straightforward to present a sentence together with a rating scale. It is a combination of the `FlashSentence` and `Message` controllers.

Option	Default	Description
s	obligatory	The sentence.
q	obligatory	The question.
as	obligatory	Answers (as for `Question`).
instructions	`null`	As for `Question`
hasCorrect	`false`	As for `Question`
autoFirstChar	`false`	As for `Question`
showNumbers	`true`	As for `Question`
randomOrder	as for `Question`	As for `Question`
presentAsScale	`false`	As for `Question`
leftComment	`null`	As for `Question`
rightComment	`null`	As for `Question`
timeout	`null`	As for `Question`

Results

Each `AcceptabilityJudgment` adds **two** lines to the results file. The first line is the same as for the `FlashSentence` controller; the second line is the same as for the `Question` controller.

DashedAcceptabilityJudgment

This is like `AcceptabilityJudgment`, but it uses `DashedSentence` instead of `FlashSentence`. This is useful for running speeded acceptability judgment tasks (which otherwise require some rather creative use of shuffle sequences).

Option	Default	Description
s	obligatory	The sentence.
q	obligatory	The question.
as	obligatory	Answers (as for `Question`).
instructions	`null`	As for `Question`
hasCorrect	`false`	As for `Question`
autoFirstChar	`false`	As for `Question`
showNumbers	`true`	As for `Question`
randomOrder	as for `Question`	As for `Question`
presentAsScale	`false`	As for `Question`
leftComment	`null`	As for `Question`
rightComment	`null`	As for `Question`
timeout	`null`	As for `Question`
mode	`"speeded acceptability"`	As for `DashedSentence` (but with different default)
display	`"dashed"`	As for `DashedSentence`
blankText	`"\u2014\u2014"`	As for `DahsedSentence`
wordTime	`300`	As for `DashedSentence`
wordPauseTime	`100`	As for `DashedSentence`
sentenceDescType	`"literal"`	As for `DashedSentence`
showAhead	`true`	As for `DashedSentence`
showBehind	`true`	As for `DashedSentence`

Results

Each `DashedAcceptabilityJudgment` adds **two** lines to the results file if the "mode" option is set to "speeded acceptability". If it is set to "self-paced reading", the number of lines is 1 + the number of words in the sentence. The first line(s) are the same as for the `DashedSentence` controller; the last line is the same as for the `Question` controller.

VBox

The `VBox` controller makes it possible to combine multiple controllers to form a single controller. Each controller in the VBox is displayed at the same time, one on top of the other. This allows the functionality of simple controllers to be reused in the construction of more complex controllers. For an example of a controller constructed using a VBox, see `js_includes/acceptability_judgment.js`.

Currently, you canno create VBox controllers inline in your data file. Instead, you must create a new controller which passes its `options` dictionary to the VBox. E.g., by placing something like the following at the beginning of your data file:

```
define_ibex_controller({
    name: "MyController",

    jqueryWidget: {
        _init: function () {
            this.options.transfer = null; // Remove 'click to continue message'.
            this.element.VBox({
```

```
                options: this.options,
                triggers: [1],
                children: [
                    "Message", this.options,
                    "AcceptabilityJudgment", this.options,
                ]
            });
        }
    },

    properties: { }
});
```

Options

Option	Default	Description
children	obligatory	An array of child controllers. Has exactly the same format as an array of elements for an item
triggers	obligatory	In order to determine when a VBox element is complete, you must give an array of the indices of those of its children which are "triggers" (indices start from 0). When each of the trigger elements is complete, the VBox is complete.
padding	`"2em"`	The amount of vertical padding to place between the children. This should be a CSS dimension.
vboxCallbackWhenChildFinishes		If set, this function is called every time one of the VBox's children finishes. It is called with two paramaters: the index of the child, and an array of arrays (= array of lines) giving the results for that child.

Results

The `VBox` controller simply concatenates the results of its children in the order that the children were given.

Form

The `Form` controller collects data from an HTML form presented to the participant. The form may contain any combination of text boxes (`<input type="text">`), text areas (`<textarea>`), checkboxes (`<input type="checkbox">`) or radio buttons (`<input type="radio">`). It isn't necessary to wrap the `<input>` tags within a `<form>` tag; nor should you include a submit button.

You can indicate that a text area or radio button group is obligatory by adding the "obligatory" class to it (e.g. `<input type="text" class="obligatory">`). For groups of radio buttons, you need only add the "obligatory" class to one button in the group. (As usual, radio buttons in a single group should all have the same 'name' attribute.)

The "obligatory" class can also be used to indicate that a checkbox must be checked before the user can continue. (This is useful for consent checkboxes.)

By default, failure to fill in an obligatory field is signaled by an alert dialog. However, you can indicate that the error message is to be displayed on the page by inserting an empty label tag in your form:

```
<label class="error" for="FIELD_NAME"></label>
```

If you want all error messages to be displayed in the same location, add a `<label>` with 'for' set to "ALL_FIELDS".

Options

Option	Default	Description
html	obligatory	As for the `Message` controller.
continueOnReturn	`false`	If `true`, the user can complete the form by pressing the return key when a single-line text field is focused.
continueMessage	"Click here to continue"	If set to `null`, no continue message is displayed.
checkedValue	`"yes"`	The value stored in the results file when a checkbox is checked.
uncheckedValue	`"no"`	The value stored in the results file when a checkbox is unchecked.
validators	`{ }`	A JavaScript dictionary mapping field names to validating functions. Functions take the field value as their sole argument, and return `true` if the value is valid or an error string otherwise.
saveReactionTime	`false`	If `true`, controller saves the time elapsed between the form's being displayed and the participant's completing it. Reaction time appears in the results file as the value of a special `_REACTION_TIME_` field.
errorCSSClass	error	The CSS class used to indicate that a `<label>` tag is a placeholder for error messages.
obligatoryErrorGenerator		A function from field names to error messages (used when an obligatory text box or text area is not filled in).
obligatoryCheckboxErrorGenerator		A function from field names to error messages (used when an obligatory checkbox is not checked).
obligatoryRadioErrorGenerator		As above but for radio button groups.

Results

For each `<input>` or `<textarea>` tag, there is a line in the results file with the following two columns:

Column	Description
1	The name of the field (taken from the 'name' attribute of the `<input>` or `<textarea>`).
2	The answer given by the participant.

Further configuration

Miscellaneous options

You can set the values of the following variables in your data file (e.g. `var showProgressBar = true;`).

Option	Default	Description
showProgressBar	`true`	Whether or not to show a progress bar.
progressBarText	`progress`	The text that appears below the progress bar.
pageTitle	`"experiment"`	Page title.
loadingFatalErrorMessage	"	The message shown when a fatal error is encountered while loading chunks
loadingNonfatalErrorMessage	"	The message shown when a non-fatal error is encountered while loading chunks
sendingResultsMessage	. . "	The message shown while results are being sent to the server.
completionMessage	" . . . "	The message shown when the results are successfully sent to the server.
completionErrorMessage	. "	The message shown when there is an error sending the results to the server.
practiceItemTypes	`[]`	A list of types for practice sentences. A sentence whose type is in this list will have a blue "practice" heading above it.

Option	Default	Description
practiceItemMessage	`"Practice"`	The message displayed for practice items
showOverview	false	See "Overviews" section
centerItems	true	Whether or not items should be centered on the page.
equalGroupSizes	false	If true, groups in latin square designs are required to contain equal numbers of items.
manualSendResults	false	See "Sending Results Early" subsection of "Shuffle Sequences" section.

Configuring `js_includes`, `data_includes` and `css_includes`

The directories `js_includes`, `data_includes` and `css_includes` directories contain JavaScript and CSS files that are necessary for the running of an experiment. The most important of these is the data file in `data_includes` containing the list of items for the experiment, but each controller also has its own file in `js_includes` (for example, the `DashedSentence` controller lives in `dashed_sentence.js`). Many controllers also define some CSS classes in corresponding files in `css_includes` (e.g. `DashedSentence.css`). If you write your own controllers, you need to put the JavaScript and CSS files in these directories.

When the webpage for an experiment is accessed, the server concatenates all the files in `js_includes` and serves them up as a single file (ditto for `data_includes` and `css_includes`). You can tell the server to ignore some of the files in `js_includes` and `css_includes` by editing the variables `JS_INCLUDES_LIST`, `DATA_INCLUDES_LIST` and `CSS_INCLUDES_LIST` in `server_conf.py`. (There is a comment documenting how to do this.) You may wish to exclude JavaScript and CSS files which are not used by your experiment in order to reduce the size of the file that needs to be downloaded. Since the files in both directories are named after the controllers with which they are associated, it is easy to see which files are superfluous.

Important: the file `global_main.css` is required by all controllers.

HTML Code

Ibex provides three methods of passing HTML code to the `Message` and `Form` controllers (currently the only controllers which have an `html` option).

Method 1: HTML in a JavaScript String

Pass a JavaScript string containing the HTML code.

Method 2: HTML as a JavaScript data structre

Pass a JavaScript data structure representing the HTML Code. For example, here is the representation of a `div` element containing two paragraphs:

```
["div",
    ["p", "This is the first paragraph."],
    ["p", "This is the second paragraph.", "Containing two text nodes."]
]
```

Note that a space will automatically be inserted between the two text nodes in the second paragraph. If you wanted to set the foreground color of the `div` to red, you could use the following code:

```
[["div", {style: "color: red;"}],
    ["p", "This is the first paragraph."],
    ["p", "This is the second paragraph.", "Containing two text nodes."]
]
```

If you want to set DOM properties directly, you can use a key beginning with "@":

```
[["div", {"@style.color": "red"}],
    ["p", "This is the first paragraph."],
    ["p", "This is the second paragraph.", "Containing two text nodes."]
]
```

Elements (e.g. `&ldquo;` – a left double quote) are specified as in the following example:

```
[["div", {"@style.color": "red"}],
    ["p", "This is the first paragraph."],
    ["p", "This is the second paragraph.", "Containing two text nodes."],
    ["p", "Here's a paragraph where ", ["&ldquo;"], "this", ["&rdquo;"], " is quoted."]
]
```

Spaces are not automatically inserted before and after elements.

Method 3: HTML in a separate file

Place the HTML in a file in the `chunk_includes` dir. Files in this directory can be included as follows:

```
{ html: { include: "file_name.html" } }
```

Creating your own controllers

This is quite easy if you are familiar with JavaScript/DHTML and the jQuery JavaScript library. (One of the changes in Ibex from webspr 0.2 is that controllers are now JQuery.ui widgets.) As an example, here's the code for a simplified version of the `Message` controller:

```
define_ibex_controller({
name: "Message",

jqueryWidget: {
    _init: function() {
        // Boilerplate code that appears in all controllers.
        this.cssPrefix = this.options._cssPrefix;
        this.utils = this.options._utils;
        this.finishedCallback = this.options._finishedCallback;

        this.html = this.options.html;
        this.element.addClass(this.cssPrefix + "message");
        this.element.append(htmlCodeToDOM(this.html));

        this.transfer = dget(this.options, "transfer", "keypress");
        assert(this.transfer == "keypress" || typeof(this.transfer) == "number",
```

```
                "Value of 'transfer' option of Message must either be the string 'keypress' or a number")

        if (this.transfer == "keypress") {
            var t = this;
            // See below for info on 'safeBind'.
            this.safeBind($(document), 'keydown', function () {
                t.finishedCallback(null);
                return true;
            });
        }
        else {
            // See below for info on 'this.utils'.
            this.utils.setTimeout(this.finishedCallback, this.transfer);
        }
    }
},

properties: {
    obligatory: ["html"],
    countsForProgressBar: false,
    htmlDescription: function (opts) {
        var d = htmlCodeToDOM(opts.html);
        return truncateHTML(d, 100);
    }
}
});
```

The call to `ibex_controller_set_properties` sets some properties of the `Message` controller. The `obligatory` option specifies those options which must obligatorily be given to the controller. In this case, it is obligatory that the controller be given an `"html"` option. The `obligatory` key must be present in the object passed to `ibex_controller_set_properties`. The `countsForProgressBar` property is optional and is `true` by default. It determines whether instances of the controller count towards the size of the progress bar. (This option can be overridden on an item-by-item basis by setting the `countsForProgressBar` property of an item.) The `htmlDescription` function should return either HTML or a DOM node which gives a brief summary of the content of an instance of the controller for use in overviews.

The `options` dictionary for the widget has three special values set:

- A function (`_finishedCallback`) which should be called with lines to be added to the results file when the controller is complete.
- A `_utils` object which contains some useful functions.
- A `_cssPrefix` string, which gives the controller its designated CSS prefix for all CSS class names/ids which it adds to DOM nodes (see section below, "CSS mangling").

In the case of `Message`, `_finishedCallback` is called with `null` as its argument
because this controller does not add any lines to the results file.
In general, the format of a non-null argument to `finishedCallback` is as follows:

```
[
    // Line 1.
    [ ["fieldname1", value1], ["fieldname2", value2], ["fieldname3", value3], ... ],
    // Line 2.
    [ ["fieldname2", value2], ["fieldname2", value2], ["fieldname3", value3], ... ],
    ...
]
```

As can be seen in the code for `Message`, the `utils` object provides a `setTimeout` method similar to the builtin `setTimeout` function of JavaScript. Any timeouts set using this method are automatically cleared when the controller is complete. Similarly, a `safeBind` method has been added to jQuery, which ensures that event handlers are automatically unregistered once a controller instance is finished.

CSS mangling

Every class/id name in a CSS file in the `css_includes` directory is prepended with the string 'FILENAME-'. The CSS for each controller should live in `CONTROLLER_NAME.css`. This ensures that there are no namespace clashes between controllers. Note that if you wish to prevent automatic CSS mangling (e.g. because of a bug in the mangling code that you need to work around), you may prefix your CSS file name with `"global_"`. This causes the file to be served as-is.

Overviews

Sometimes it's useful to get an overview of the sequence of items in an experiment without actually running through each item. There are two ways of getting websprr to display an overview of this sort. The first is to add the statement `var showOverview = true;` to your data file. The second is to go to the page `overview.html` instead of `experiment.html`.

Cross-browser compatibility

I have tested compatibility with the following browsers:

- Internet Explorer 6 and 7. (Note that support for IE 5 has been dropped as of version 0.3 of this software, since the jQuery library doesn't support this browser.)
- Firefox 1, 1.5, 2, and 3.
- Safari 3.
- Opera 9.
- Google Chrome.

Known cross-browser issues:

- Overviews (see previous section) currently do not work on all versions of Internet Explorer.
- Pressing '6' and '7' in Opera affects text size; this interacts badly with acceptability judgments on scales including 6 and 7. I have not yet found a way of preventing Opera from interpreting these keypresses.
- The stand-alone server serves up the JavaScript and CSS include files with a `Pragma: nocache` so that any changes you make will be immediately reflected if you refresh `experiment.html`. However, some versions of Internet Explorer ignore this pragma (I think this is a bug in IE, but not 100% sure yet), so you will need to delete your temporary internet files and then refresh. This is not such a big issue for live experiments, but it makes developing and testing experiments using IE a big PITA. The obvious workaround is simply not to use IE for these purposes.

Terminological clarifications

Unfortunately, the terms I've used relating to the design of experiments (latin squares, etc.) are confusing as they use some non-standard terms, and make non-standard use of some standard terms. Specifically:

- The term ‘item number’ is used to describe the number assigned to a controller based on its position in the list of controllers in the data file. Of course, controllers are not normally in one-to-one correspondence with ‘items’ in the usual sense of the term.
- The term ‘group’ is used to refer to what are usually called items (i.e. sets of conditions).

Known problems and issues

The following are some subtle problems which can often arise when writing a data file:

- A missing comma in the list of sentences in data.js can cause highly obscure JavaScript errors with no obvious relation to their source.
- Some browsers accept trailing commas in JavaScript array literals (i.e. they accept [1,2,3,4,] as a fine array); others do not. If your browser of choice accepts array literals of this form, be sure to check that your data.js has no trailing commas so that there will be no browser incompatibilities. It is quite easy to introduce trailing commas by accident if you comment out some of the items in the `items` array.

Most browsers allow strings to be indexed using square brackets (i.e. `"foo""[1] == "f"`). Internet Explorer, however, requires the use of the string’s `charAt` method.

For debugging, I recommend using Firefox’s JavaScript console. Most syntax errors in a data file will result in an alert popping up with a warning that the `items` array has not been defined. You can usually get a much more informative error message by looking at the JavaScript console.

Changes

Changes between 0.3.8 (current version and 0.3.7):

- Fix bug with VBox controller which caused transition to next item to fail sometimes.
- Fix bug in `Form` controller which prevented the `continueOnReturn` option

  from taking effect (thanks to Laurel Brehm).
- Some preliminary support for playing sounds (not yet documented).

Changes between 0.3.7 (current version) and 0.3.6:

- Fix bug which prevents `countsForProgressBar` option taking effect.

Changes between 0.3.6 and 0.3.5:

- Improve loading of files in chunk_includes. Chunks are now preloaded so that experiments will not be disrupted.

Changes between 0.3.5 and 0.3.4:

- Fix a bug in the recording of answers for certain options of the `Question` controller (thanks to Michael Yoshitaka Erlewine).

- Fix a bug in VBox which caused errors to appear on the console in some cases (thanks to Alexandre Cremers).

Changes between 0.3.4 (current version) and 0.3.3:

- Fix a bug which caused commas to be unescaped in 'Form' controller results.
- Fix a minor bug in server.py (most likely did not have any user-visible effects).

Changes between 0.3.3 and 0.3.2:

- Improve recording of HTML sentences in results file with `AcceptabilityJudgment` and `FlashSentence`.
- Fix a venerable bug in server.py which caused files to be served with a leading blank line with the toy server.
- Remove 'md5' option for 'sentenceDescType'.

Changes between 0.3.2 and 0.3.1:

- Corrected error in docs ("requiresConsent" to "consentRequired").
- Made it possible to use HTML with `FlashSentence` controller (and hence also with `AcceptabilityJudgment`). Currently undocumented; see Google group discussion.

Changes between 0.3.1 and 0.3.0:

- Fixed a regression introduced in 0.3.0 (trailing comma stopped experiments working in IE).

Changes between 0.3.0 and 0.3-beta19:

- Fixed a bug involving timeouts with the `AcceptabilityJudgment` controller.

Changes between 0.3-beta19 and 0.3-beta18:

- Added special `__SetCounter__` controller (see comment in `example_data.js`).

Changes between 0.3-beta18 and 0.3-beta17:

- Add option to save reaction times to `Form` controller.

Changes between 0.3-beta17 and 0.3-beta16:

- Fix a bug which caused timings for question answers to be recorded incorrectly in some instances.
- Add the `presentHorizontally` option to the `Question` controller.
- Add the option to assign keys to answers based on the (random) order in which they are presented.

Changes between 0.3-beta16 and 0.3-beta15:

- Fix bug which stopped ibex working on Internet Explorer.

Changes between 0.3-beta15 and 0.3-beta14:

- Added the `DashedAcceptabilityJudgment` controller for easy speeded acceptability judgments.

- Cleaned up the way controllers are defined (not a user-visible change).
- Fixed a bug in the file-locking code in `server.py`.
- Fixed a bug with result concatenation in the `VBox` controller which surface on Chrome/Safari.
- Make it possible to create `Form` and `Message` controllers which can't be completed (useful in VBoxes).

Changes between 0.3-beta14 and 0.3-beta13:

- It is now possible to set the `countsForProgressBar` option at the level of individual items.
- Add 'continueOnReturn' option to the `Form` controller.
- Add the option to display all words at the center of the screen with the `DashedSentence` controller.
- Automatic generation of `experiment.html` and `overview.html` can now deal with installations where `sever.py` has been renamed.
- Upgrade to jquery 1.5.1 and jquery-ui 1.8.10
- Minor bug fixes.

Changes between 0.3-beta13 and 0.3-beta12:

- Ibex no longer depends on cookies to set the value of the latin square counter.
- Trailing comma removed in example_data.js.
- `DashedSentence` controller now uses spans instead of divs for the individual words. This should make things work more smoothly with right-to-left languages (though I haven't tested this yet).
- Highlighting text no longer shows hidden words when using `DashedSentence`.
- Fixed regression introduced in 0.3 where line breaks in SPR sentences did not get recorded in the results.

Changes between 0.3-beta12 and 0.3-beta11:

- Fix a minor bug which caused 'DYNAMIC' to be used instead of 'NULL' in some cases in the results file.
- Fixed a bug which occasionally led to inconsistent values for the unique user ID in the results file (second column in each line).

Changes between 0.3-beta11 and 0.3-beta10:

- Added the `modifyRunningOrder` config option.
- `example_data.js` has been slimmed down and reformatted.

Changes between 0.3-beta10 and 0.3-beta9:

- Time is now recorded for the last word in a sentence in self-paced reading experiments.

Changes between 0.3-beta9 and 0.3-beta8:

- It is now possible to control the point in the experiment at which results are sent to the server using the special `__SendResults__` controller.
- Upgrade from jQuery 1.4.0 to 1.4.2.
- Fix bug which prevented "keypress" transfer working with the `Separator` controller.
- Fix bug which prevented the `DashedSentence` controller from being able to display text one "chunk" at a time.
- Minor fixes.

Changes between 0.3-beta8 and 0.3-beta7:

- Fix bug which caused self-paced reading experiments to display incorrectly in IE 8.
- Fix bug which caused question answers to appear without numbers in IE.

Changes between 0.3-beta7 and 0.3-beta6:

- Fix bug which caused questions not to be recorded in results file.
- Checkboxes can be specified as obligatory in the `Form` controller.
- Minor changes to example data file.

Changes between 0.3-beta6 and 0.3-beta5:

- Fix bug in VBox.js which caused some things to appear off center. (This bug only affected the presentation of items, not the results.)

Changes between 0.3-beta5 and 0.3-beta4:

- Improvements to the recording of times in self-paced reading experiments. Times should now be recorded a little more accurately.
- DashedSentence controller works on the iPhone (though it's not practical for actually doing experiments).
- Option to minify JavaScript code in `data_includes` dir (turned off by default).
- In the results file, the MD5 hash of the subject's IP address has been replaced with a hash based on the IP address together with a variety of other identifying information (e.g. user agent string, subject's screen size). This both provides a better unique identifier, and is preferable from a privacy point of view. (It isn't difficult to reverse an IP address hash, but it would be pretty difficult to reverse the new hashes.)

Changes between 0.3-beta4 and 0.3-beta3:

- Fix a bug in the `Form` controller that caused values for radio buttons to be incorrectly recorded in the results file.

Changes between 0.3-beta3 and 0.3-beta2:

- `Form` controller added for collecting data from participants.
- New method of passing HTML to the `Message` and `Form` controllers (HTML files can be included from `chunk_includes` dir).

Changes between 0.3-beta2 and 0.3-beta1:

- Bug fixes.

Changes between 0.3-beta1 and 0.2.7:

- Name changed to "Ibex" (Internet Based EXperiments).
- Removed dependence on `paste` module for stand-alone server. The server now requires only modules that are bundled with Python >= 2.3.
- JavaScript code now uses jQuery 1.4. Controllers are jQuery.ui widgets.
- New ways of configuring the server. The server can now issue an HTTP GET request to get its config variables. Config variables can be inlined in server.py as well as being given in the separate `server_conf.py` file. These features are used for the Ibex Farm (http://spellout.net/ibexfarm).
- CSS files in css_includes now modified automatically by the server to ensure that there are no CSS namespace clashes between different widgets.
- Server now caches css and js includes (i.e. the results of requests like server.py?includes=js).
- Support for IE 5.5 dropped (jQuery does not support IE 5.5).
- Some additional options for the `DashedSentence` controller.