experimentator Documentation

Release 0.3.1

Henry S. Harrison

June 07, 2016

Contents

Documentation contents

Do you write code to run experiments? If so, you've probably had the experience of sitting down to code an experiment but getting side-tracked by all the logistics: crossing your independent variables to form conditions, repeating your conditions, randomization, storing intermediate data, etc. It's frustrating to put all that effort in before even getting to what's really unique about your experiment. Worse, it encourages bad coding practices like copy-pasting boilerplate from someone else's experiment code without understanding it.

The purpose of **experimentator** is to handle all the boring logistics of running experiments and allow you to get straight to what really interests you, whatever that may be. This package was originally intended for behavioral experiments in which human participants are interacting with a graphical interface, but there is nothing domain-specific about it–it should be useful for anyone running experiments with a computer. You might say that **experimentator** is a library for 'repeatedly calling a function while systematically varying its inputs and saving the data' (although that doesn't do it full justice).

CHAPTER 1

Not handled here

- graphics
- timing
- hardware interfacing
- statistics
- data processing

The philosophy of experimentator is to do one thing and do it well. It is meant to be used with other libraries that handle the above functionality, and gives you the freedom to choose which you prefer. It is best suited for someone with programming experience and some knowledge of the Python ecosystem, who would rather choose the best tool for each aspect of a project than use an all-in-one package.

Of course, there are alternatives that offer experimental design features along with other capabilities. A selection, as well as recommended complimentary packages, are listed later in the documentation.

CHAPTER 2

An example

To demonstrate, let's create a simple perceptual experiment. For the sake of example, imagine we will present some stimulus to either the left or right side of the screen for a specified amount of time, and ask the participant to identify it. We'll use a factorial 2 (side) x 3 (display time) design, and have a total of 60 trials per participant (10 per condition). Here's how it might look in experimentator:

```python
import random
from time import time
from experimentator import Experiment, order


def present_stimulus_and_get_response(stimulus, side, duration):
    # Use your imagination...
    return random.choice(['yes', 'no'])


def run_trial(experiment, trial):
    stimulus, answer = random.choice(
        list(experiment.experiment_data['stimuli'].items()))
    start_time = time()
    response = present_stimulus_and_get_response(trial.data['side'], trial.data['display_time'])
    result = {
        'reaction_time': time() - start_time,
        'correct': response == answer
    }
    return result


if __name__ == '__main__':
    independent_variables = {
        'side': ['left', 'right'],
        'display_time': [0.1, 0.55, 1],
    }
    stimuli_and_answers = {
        'cat.jpg': 'yes',
        'dog.jpg': 'no',
    }

    experiment = Experiment.within_subjects(independent_variables,
                                            n_participants=20,
                                            ordering=order.Shuffle(10),
                                            filename='exp_1.exp')

    experiment.experiment_data['stimuli'] = stimuli_and_answers
    experiment.add_callback('trial', run_trial)
    experiment.save()
```

Running this script will create the experiment in the file `exp_1.exp`. We can now run sessions from the command line:

```
exp run exp_1.exp participant 1
# or
exp run exp_1.exp --next participant
```

Eventually, we can export the data to a text file:

```
exp export exp_1.exp exp_1_data.csv
```

Or, access the data in a Python session:

```
from experimentator import Experiment

data = Experiment.load('exp_1.exp').dataframe
```

In this example the data will be a pandas `DataFrame` with six columns: two index columns with labels `'participant'` and `'trial'`, two columns from the IVs, with labels `'side'` and `'display_time'`, and two data columns with labels `'reaction_time'` and `'correct'` (the keys in the dictionary returned by `run_Trial`).

CHAPTER 3

Installation

Note: If you use experimentator in your work, published or not, please let me know. I'm curious to know what you use it for! If you do publish, citation information can be found here.

3.1 Dependencies

Experimentator requires Python 3.3 or later. It also depends on the following Python libraries:

- numpy
- pandas
- docopt
- schema
- PyYAML
- NetworkX

Required for tests:

- pytest

Required for generating docs:

- Sphinx
- numpydoc
- sphinx-rtd-theme

The easiest way to install these libraries, especially on Windows, is with Continuum's free Python distribution Anaconda. For experimentator, Anaconda3 or the lightweight Miniconda3 is recommended, although you can create a Python3 `conda` environment regardless of which version you initially download.

For example, to install dependencies to a clean environment (with name `experiment`):

```
conda update conda
conda create -n experiment python=3 pip
source activate experiment
conda install numpy pandas pyyaml networkx
pip install docopt schema
```

3.2 From PyPi

To install (and upgrade) experimentator:

```
pip install --upgrade experimentator
```

Be sure to run `pip` from a Python 3 environment.

3.3 From source (development version)

Experimentator is hosted on GitHub:

```
git clone git@github.com:hsharrison/experimentator
cd experimentator
pip install -e . --upgrade
```

CHAPTER 4

Other libraries

Please, feel free to submit a pull request to add your software to one of these lists.

4.1 Alternatives

The Python ecosystem offers some wonderful alternatives that provide experiment logistics in addition to other functionality like graphics and input/output:

- expyriment: Graphics, input/output, hardware interfacing, data preprocessing, experimental design. If you are coming from the Matlab world, this is the closest thing to Psychtoolbox.
- OpenSesame: An all-in-one package with a graphical interface to boot. An impressive piece of software.

4.2 Complimentary libraries

What about all those important things that experimentator doesn't do? Here's a short selection. If you're already using Python some of these will go without saying, but they're included here for completeness:

- ***experimental design***
  - pyDOE: Construct design matrices in a format that experimentator can use to build your experiment.
- ***graphics***
  - PsychoPy: A stimulus-presentation library with an emphasis on calibration and temporal precision. Unfortunately, at the time of this writing it is not yet Python3-compatible, and so cannot be easily combined with experimentator.
  - Pygame: Very popular.
  - Pyglet: A smaller community than Pygame, but has several advantages, including cross-compatibility and a more pythonic API. Includes OpenGL bindings.
  - PyOpenGL: If all you need is to make OpenGL calls.
- ***graphical user interfaces***
  - urwid: Console user interface library, ncurses-style.
  - wxPython: Python bindings for the wxWidgets C++ library.
  - PyQT: QT bindings.
  - PySide: Another QT option.

    - PyGTK: Python bindings for GTK+.

- ***statistics and data processing***
    - pandas: Convenient data structures. Experimental data in experimentator is stored in a pandas `DataFrame`.
    - numpy: Matrix operations. The core of the Python scientific computing stack.
    - SciPy: A comprehensive scientific computing library spanning many domains.
    - Statsmodels: Statistical modeling and hypothesis testing.
    - scikit-learn: Machine learning.
    - rpy2: Call `R` from Python. Because sometimes the model or test you need isn't in statsmodels or scikit-learn.

License

Licensed under the MIT license.

5.1 Contents

5.1.1 Concepts

About this documentation

This documentation aims to be comprehensive, but be aware that there is also rich information available in docstrings. These can be accessed at the interactive prompt with the `help` function; they are also reproduced in *API reference*.

Experiment structure

In experimentator, experiments (represented by an *`Experiment`* instance) are organized as hierarchical tree structures. Each section of the experiment (represented by an *`ExperimentSection`* instance) is a node in this tree, and its children are the sections it contains. Levels on the tree are named; common level names in behavioral research are `'participant'`, `'session'`, `'block'`, and `'trial'`. For example, an experiment with two participants with two blocks of three trials each would have a tree that looks like this:

```
'_base'                       ______________1______________
                             /                             \
'participant'         ______1______                   ______2______
                     /             \                 /             \
'block'         ___1___         ___2___         ___1___         ___2___
               /   |   \       /   |   \       /   |   \       /   |   \
'trial'        1    2    3     1    2    3     1    2    3     1    2    3
```

The top level is always called `'_base'`; the leading underscore indicates that you should not have to refer to this level directly. All other level names are arbitrary and are specified when the experiment is created.

An important principle of experimentator is that each section only handles its children, the sections immediately below it. In a structure with levels `'participant'`, `'block'`, and `'trial'`, every block section knows how to create and order trials (e.g., by crossing *independent variables*), but knows nothing of participants. Likewise, every participant section organizes the blocks under it, but lets each block figure out its constitutent trials. The only exception to this rule is in the case of *non-atomic orderings*.

Note: For simplicity, this documentation uses the term *trial* to mean the lowest level of an experiment, even though experimentator will let you use whatever string you want to name this level.

Navigating structure

Note: Be aware that experimentator uses 1-based indexing when numbering sections and indexing `ExperimentSection` instances, as in the diagram above.

An experimental hierarchy can be explored in a number of ways. Given an `Experiment` object, any section can be found by direct indexing:

```
# Assuming the same structure as the diagram above.
experiment[1]        # first participant
experiment[1][2]     # second block of first participant
experiment[1][2][2]  # second trial of second block of first participant
experiment[1, 2, 2]  # same as previous
```

Alternatively, the `subsection` method can be used. The following finds the same sections as the previous example:

```
experiment.subsection(participant=1)
experiment.subsection(participant=1, block=2)
experiment.subsection(participant=1, block=2, trial=2)
```

The generator method `all_subsections` yields all subsections matching the given criteria. For example, with the same experiment structure,

```
list(experiment.all_subsections(block=2, trial=1))
```

will return the same list as

```
[experiment.subsection(participant=1, block=2, trial=1),
 experiment.subsection(participant=2, block=2, trial=1)]
```

There are other methods to help find specific sections, for example `find_first_not_run`, `find_first_partially_run`, and the more general `depth_first_search` and `breadth_first_search`. These last two methods allow you to define the search criteria with a custom `key` function that returns `True` for the desired section.

Design

In experimentator, every section has a *design*, represented by a `Design` object (usually, these will be created for you). Most of the time, all sections at the same level have the same design (but see *Heterogeneous experiment structures*). The design is a high-level description of one level of an experiment. It includes everything experimentator needs to know to create the children of a section. This consists of two things: *independent variables* and an *ordering method*.

An experiment requires multiple `Design` instances in a certain relationship to each other. Such a collection is modeled with `DesignTree` objects. Again, you usually will not manually create these.

Independent variables

A central concept in experimentator (and in experimental design more generally) is that of *independent variables*, or IVs. An IV is a variable that you are explicitly varying in order to test its effects. The easiest way to represent IVs in experimentator is using a dictionary. Each key is a string, the name of an IV. Each value is either a list, representing the possible values the IV can take, or `None` if the IV takes continuous values (continuous values are only possible with a *design matrix*). For example:

```
>>> independent_variables = {
...     'congruent': [True, False],
...     'distractor': [None, 'left', 'right'],
... }
```

Note: In Python, dictionaries have no order. In most cases, the order of IVs is not important and so representing IVs as dictionaries will work fine. However, there are times when the order you specify the IVs is important. This is the case, for example, when using a *design matrix*, because each column of the design matrix refers to one IV. You will need to rely on the order of IVs in order to know which column controls which IV. In these cases you should use one of two alternative ways of representing IVs: using a `collections.OrderedDict`, or a list of 2-tuples. Here is an example of the latter method (equivalent to the previous example):

```
>>> independent_variables = [
...     ('congruent', [True, False]),
...     ('distractor', [None, 'left', 'right']),
... ]
```

When you specify your IVs, you will specify them separately for every level of the experiment. That is, every IV is associated with a level of the experimental hierarchy. This determines how often the IV value changes. For example, a within-subjects experiment will probably have IVs at the `'trial'` level, a between-subjects experiment will have IVs at the `'participant'` level, and a mixed-design experiment will have both. An IV at the `'participant'` level will always take the same value within each participant. Similarly, a blocked experiment may have IVs at the `'block'` level; these IVs will only take on a new value when a new block is reached.

IV values are ultimately passed to your *run callback* as a *condition*. A condition is a combination of specific IV values. Although you don't need to create conditions yourself, you can think of them as dictionaries mapping IV names to values. For example, the six conditions generated by a full factorial cross of the IVs above are:

```
[{'congruent': True, 'distractor': None},
 {'congruent': True, 'distractor': 'left'},
 {'congruent': True, 'distractor': 'right'},
 {'congruent': False, 'distractor': None},
 {'congruent': False, 'distractor': 'left'},
 {'congruent': False, 'distractor': 'right'}]
```

Just like IVs, different conditions apply at different levels of the experimental hierarchy. These conditions propagate down the tree. For example, imagine a trial has one of the conditions in the list above, `{'congruent': True, 'distractor': None}`. The block that the trial is part of may have an additional condition, like `{'practice': False}`. When the trial is run, these conditions are effectively merged.

Note: This merging is implemented with the standard-library object `collections.ChainMap`. A `ChainMap` can be accessed just like a dictionary; this is the sense in which it is correct to say that the conditions are merged. To continue the example, one can access the IV values without worrying about what level each IV came from:

```
>>> condition['congruent']
True
>>> condition['practice']
False
```

However, it is possible to differentiate the conditions if needed, using the `maps` attribute. See the `ChainMap` docs for details. You might see something like this:

```
>>> condition.maps[0]
{'trial': 1,
 'congruent': True,
 'distractor': None}
>>> condition.maps[1]
{'block': 2,
 'practice': False}
>>> condition.maps[2]
{'participant': 1}
```

Orderings

The second element of a *design* is an *ordering method*. The ordering method determines how children of a section wll be ordered (and possibly repeated). For example, an experiment may shuffle trials within each block, counter-balance blocks within each session, and put all sessions within each participant in the same order.

Each ordering method is a class in the `experimentator.order` module. Currently, experimentator includes `Ordering` (the base class, resulting in a deterministic order), `Shuffle`, `CompleteCounterbalance`, `Sorted`, and `LatinSquare`. `Shuffle` is usually the default, except if you're using a *design matrix*, in which case experimentator assumes you want a deterministic order and makes `Ordering` the default.

Each ordering method class has different parameters, so see the specific API reference for details. Commonly, the first argument is `number`, which specifies the number of times each condition will be repeated. For example, with the ordering method `Shuffle(3)`, each unique condition will be repeated three times, and then the order will be randomized.

Non-atomic orderings The included ordering classes can be divided into two categories: atomic and non-atomic. If every ordering of sections is independent of all other orderings, then the ordering method is atomic. For example, if trials within a block are shuffled, then the ordering of trials within each block will be independent. Each block can shuffle its trials without needing to know the order of trials within the other blocks.

However, this is not the case for non-atomic orderings. The ordering of sections using non-atomic orderings are dependent on each other. For example, if blocks within a session are counterbalanced using `CompleteCounterbalance`, then each session cannot, on its own, determine the order of blocks within it.

Non-atomic orderings are implemented by automatically creating a new independent variable. For example, if the `'block'` level has three conditions (e.g., one IV with three possible values) and a `CompleteCounterbalance` ordering (with `number=1`), then there are six possible orderings of blocks. A new IV called `'counterbalance_order'` will be automatically created one level up (e.g., at the `'session'` level), with six possible values (the integers 0-5).

Don't forget to take this automatically-created IV into account when designing your experiment. In the above example, if there are no other IVs at the `'session'` level, and `number=1` for the `'session'` ordering, there will still be six sessions per participant due to the six conditions defined by the `'counterbalance_order'` IV.

Only `Ordering` and `Shuffle` are atomic; the other ordering methods provided in experimentator are non-atomic (the `Sorted` ordering method straddles the line; it may or may not be atomic, depending on the parameter `order`. If `order='ascending'` or `order='descending'`, then the ordering method is atomic as it is sorted the same way at every section. However, if `order='both'`, then it is non-atomic and a new IV `{'order': ['ascending', 'descending']}` will be created).

Why use levels?

You may be wondering how many levels to use, or why to use them at all (after all, flat is better than nested). That decision must be made on a case-by-case basis. For example, imagine your experiment has sessions of 20 trials, divided into two blocks. As long as the order of conditions within each session is correctly specified (for example, by using a *design matrix*), using an explicit `'block'` level may not be necessary. Alternatively, you could define a `'block'` level but not a `'trial'` level and stick a trial loop inside the block. However, using levels makes it possible to...

- associate an IV with a level, facilitating the creation and ordering of conditions.
- run code before and/or after every section at a particular level, using *section context managers*. For example, offer participants a break between blocks.
- run experiment sections by level (using the *command-line interface*). For example, using blocks you could do

```
exp run my_exp.exp participant 1 block 2
```

  rather than the more awkward

```
exp run my_exp.exp participant 1 --from 11
```

- index the data by level, after running the experiment, using hierarchical indexing. For example, to get the third trial of the first participant's second block you could do

```
experiment.dataframe.loc[(1, 2, 3), :]
```

  or to get the first trial of the second block of every participant,

```
data.xs((2, 1), level=('block', 'trial'))
```

Heterogeneous experiment structures

A final concept to explain is the difference between homogeneous and heterogeneous experiment structures. In a homogeneous experiment, every section at the same level has the same *design*. For example, if the first block contains ten trials and the second block contains twenty, the experiment structure is heterogeneous. If the order of blocks within the first session is random but the order of blocks within the second session is counterbalanced, the experiment structure is heterogeneous. Even different possible IV values across sections is enough to break homogeneity.

Heterogeneous experiments are a little trickier to set up, but they are fully supported by experimentator. See *Constructing heterogeneous experiments*.

5.1.2 Creating an experiment

The typical workflow using experimentator is relatively straightforward:

1. Create an `Experiment` instance.
2. Run the experiment using the *command-line interface*.
3. Inspect, analyze or export the resulting data.

Constructor methods

The most general way to create an `Experiment` is to use `Experiment.new`, but there are a number of other methods that may be easier for many use cases.

Simple constructor methods

These methods construct *Experiment* instances based on common experimental designs.

- *Experiment.within_subjects*: Construct an experiment with levels `'participant'` and `'trial'`, and IVs only at the `'trial'` level. For example:

```
>>> from experimentator import Experiment, order
... independent_variables = {
...     'side': ['left', 'right'],
...     'display_time': [0.1, 0.55, 1],
... }
... experiment = Experiment.within_subjects(
...     independent_variables,
...     n_participants=20,
...     ordering=order.Shuffle(10)
... )
```

  The above creates a 2 (side) by 3 (display time) within-subjects experiment, with 10 trials of each condition and 20 participants. Trials will be shuffled within participants.

- *Experiment.blocked*: Construct an experiment with levels `'participant'`, `'block'`, and `'trial'`, with IVs at the `'trial'` level (and optionally at the `'block'` level also). The following constructs an experiment identical to the previous example, except with each participant's 60 trials split into two blocks:

```
>>> from experimentator import Experiment, order
... independent_variables = {
...     'side': ['left', 'right'],
...     'display_time': [0.1, 0.55, 1],
... }
>>> experiment = Experiment.blocked(
...     independent_variables,
...     n_participants=20,
...     orderings={
...         'trial': order.Shuffle(5),
...         'participant': order.Ordering(2),
...     }
... )
```

  In the above example, it doesn't matter what ordering method we use at the `'block'` level; since there are no block-level IVs, all blocks are identical. We could, alternatively, introduce an IV at the block level:

```
>>> from experimentator import Experiment, order
...  independent_variables = {
...     'side': ['left', 'right'],
...     'display_time': [0.1, 0.55, 1],
... }
>>> experiment = Experiment.blocked(
...     independent_variables,
...     block_ivs={'difficulty': ['easy', 'hard']}
...     n_participants=20,
...     orderings={'trial': order.Shuffle(5)}
... )
```

  In this example, we introduced the IV `'difficulty'` with two levels. Since we didn't specify an ordering for blocks, `Shuffle(1)` will be used. In other words, each participant will experience one `'easy'` and one `'hard'` block, in a random order.

- *Experiment.basic*: Construct an experiment with arbitrary levels but a homogeneous structure. This constructor can handle any experimental structure, with the exception of *heterogeneity*. For example, to create the same blocked experiment as in the previous example:

```
>>> from experimentator import Exeriment, order
>>> independent_variables = {
...     'trial': {
...         'side': ['left', 'right'],
...         'display_time': [0.1, 0.55, 1],
...     },
...     'block': {'difficulty': ['easy', 'hard']},
... }
>>> experiment = Experiment.basic(
...     ('participant', 'block', 'trial'),
...     independent_variables,
...     ordering_by_level={
...         'participant': order.Ordering(20),
...         'trial': order.Shuffle(5),
...     }
... )
```

  Again, the default `Shuffle(1)` will be used at the `'block'` level.

  We could also use *Experiment.basic* to make a mixed-design experiment, by adding a new IV at the `'participant'` level:

```
>>> from experimentator import Exeriment, Shuffle
>>> independent_variables = {
...     'trial': {
...         'side': ['left', 'right'],
...         'display_time': [0.1, 0.55, 1],
...     },
...     'block': {'difficulty': ['easy', 'hard']},
...     'participant': {'vision': ['monocular', 'binocular']},
... }
>>> experiment = Experiment.basic(
...     ('participant', 'block', 'trial'),
...     independent_variables,
...     ordering_by_level={
...         'participant': Shuffle(20),
...         'trial': Shuffle(5),
...     }
... )
```

  In addition to adding the IV `'vision'` at the `'participant'` level, we also changed the `'participant'` ordering from *Ordering* to *Shuffle* in order to assign participants to conditions randomly. Note that we kept the `number` parameter on the `'participant'` ordering at 20; this means our experiment will now require 40 participants, since there will be 2 conditions at the `'participant'` level.

Specification-based constructor methods

Experimentator provides a dictionary-based specification format for creating new *Experiment* instances. There are two relevant constructor methods: *Experiment.from_dict* constructs an *Experiment* given a dictionary, and *Experiment.from_yaml_file* constructs an *Experiment* given the path to a file containing the specification in YAML format.

The specification is the same for both. Central to the specification format is specifying a *DesignTree* and its constituent *Design* instances.

Design specification format See also:

Design More information on the *Design* concept.

Design.from_dict The method that implements the construction of a *Design* from a specification dictionary.

A single *Design* instance can be created from a dictionary (either a Python dict or read from a YAML file via *Experiment.from_yaml_file*). The dictionary can contain any of the following keys, all optional:

- `'name'`: The name of the level.
- `'ivs'`: The designs's independent variables. Can be a dictionary mapping IV names to possible IV values, or a list of `(name, values)` tuples. See *Independent variables*. If `'ivs'` is not specified, the design will have no IVs.
- `'order'` or `'ordering'`: The design's ordering method. Can be specified in three ways:
  - as a string, interpreted as a class name in the *order* module;
  - as a dictionary, with the key `'class'` containing the class name and the rest of the items containing keyword arguments to its constructor; or
  - as a sequence, with the first item containing the class name and the rest of the items containing positional arguments to its constructor.

  If no ordering is specified, the default is *Shuffle* (*Ordering* if a *design matrix* is used).
- `'n'` or `'number'`: The `number` argument to the specified ordering class can be specified here (or as part of the `ordering` specification).
- `'design_matrix'`: An array-like (e.g., a list of lists) specifying a design matrix to use at this level. See *Design matrices*.
- Any remaining fields are passed to the *Design* constructor as the `extra_data` argument. These values are associated with any sections created under this design. For example, you could pass `{'practice': True}` to practice blocks, to mark them as such.

For example, the following creates a *Design* instance equivalent to the one at the `'trial'` level in the previous example (of *Experiment.basic*):

```
>>> from experimentator import Design
>>> level_name, design = Design.from_dict(dict(
...     name='trial',
...     ivs={
...         'side': ['left', 'right'],
...         'display_time': [0.1, 0.55, 1],
...     },
...     ordering='Shuffle',
...     n=5,
... ))
```

For internal reasons, *Design.from_dict* outputs the level name as well as the *Design* object. This shouldn't be too important, because you will probably not be calling *Design.from_dict* directly, but rather using the dictionary format within *Experiment.from_dict* or *Experiment.from_yaml_file*.

DesignTree specification format See also:

DesignTree.from_spec The method that implements this specification.

To create an *Experiment*, multiple *Design* instances are needed, collected under a single *DesignTree*. This can also be done with a relatively simple specification format.

To create a `DesignTree` with a homogeneous structure, simply create a list of dictionaries, each specifying the `Design` of one level, ordered from top to bottom. For example, to create the `DesignTree` equivalent to the `Experiment.basic` mixed-design example above:

```
>>> from experimentator import DesignTree
>>> tree = DesignTree.from_spec([
...     dict(name='participant',
...          ivs={'vision': ['monocular', 'binocular']},
...          n=20),
...     dict(name='block',
...          ivs={'difficulty': ['easy', 'hard']}),
...     dict(name='trial',
...          ivs={
...              'side': ['left', 'right'],
...              'display_time': [0.1, 0.55, 1]},
...          n=5),
... ])
```

This example takes advantage of the default ordering of `Shuffle` for all three levels.

A `DesignTree` can also be constructed with a list of `(level_name, level_design)` tuples, though the specification format is more convenient as it can be used as part of the `Experiment` specification format.

Creating heterogeneous structures is a little more tricky; it will be described *below*.

Experiment specification format Once you can build a specification input suitable for `DesignTree.from_spec`, constructing an `Experiment` is straightforward. Create a dictionary with the following keys:

- `'design'`: The `DesignTree` spec goes here (the list of dictionaries described above). This is the only required key.
- `'file'` or `'filename'`: Use this field to associate your experiment with a data file. This is saved in the `Experiment.filename` attribute. Note that the `Experiment` will not be saved automatically; you still have to call `Experiment.save()`.
- Any remaining fields will be saved as a dictionary in `Experiment.experiment_data`. This is a good place to put local configuration that you read during a *callback*.

Using YAML All the nested lists and dictionaries required for `Experiment.from_dict` can be unwieldy. An alternative is `Experiment.from_yaml_file`, which allows you to save your specification in an external file. YAML is a file-format designed to be both human- and computer-readable.

Porting the previous mixed-design example into a YAML file would look like this:

```
design:
  - name: participant
    ivs:
      vision: [monocular, binocular]
    n: 20

  - name: block
    ivs:
      difficulty: [easy, hard]

  - name: trial
    ivs:
      side: [left, right]
      display_time: [0.1, 0.55, 1]
    n: 5

filename: mixed_experiment.exp
```

The only new piece of information here is the filename. It probably makes sense to include the filename in your YAML file, so you have a record of which data file is associated with the YAML file.

You can then create instantiate an `Experiment`, assuming the YAML above is stored in `mixed_experiment.yaml`:

```
>>> from experimentator import Experiment
>>> experiment = Experiment.from_yaml_file('mixed_experiment.yaml')
```

Note: This method is specifically for creating an `Experiment` from scratch. The data format used by `Experiment.save` for saving an in-progress experiment is also YAML, but using a different syntax, so it could be confused. This is why we recommend a different file suffix (our examples use `.exp`). The in-progress experiment file with the `.exp` suffix will still contain YAML data, but it will be less likely to be confused with the YAML file passed to `Experiment.from_yaml_file`.

Constructing an Experiment from a DesignTree

A final option for constructing an `Experiment` is to pass a `DesignTree` directly to the general constructor `Experiment.new`. For example, the following code would create the same `Experiment` as the previous example:

```
>>> from experimentator import DesignTree, Experiment
>>> tree = DesignTree.from_spec([
...     dict(name='participant',
...          ivs={'vision': ['monocular', 'binocular']},
...          n=20),
...     dict(name='block',
...          ivs={'difficulty': ['easy', 'hard']}),
...     dict(name='trial',
...          ivs={
...              'side': ['left', 'right'],
...              'display_time': [0.1, 0.55, 1]},
...          n=5),
... ])
>>> experiment = Experiment.new(tree, filename='mixed_experiment.exp')
```

Constructing heterogeneous experiments

As we've noted, constructing a *heterogenous* `Experiment` is a bit more complicated. To expand on the above example, let's imagine we want to create a two-session experiment. The first session contains only one block, with only easy trials. The second session will then contain an easy and a hard block. Furthermore, we would like to add four practice trials at the beginning of each session.

Heterogeneity is created at the level of the `DesignTree`. Remember how we built a `DesignTree` as a list of dictionaries? To create a heterogeneous `DesignTree`, we need multiple lists of dictionaries. We use a dictionary, where each value is a list of dictionaries (specifying an internally homogeneous *section* of the tree), and the keys give names to these sub-trees.

Experimentator will create the `DesignTree` by starting at the sub-tree with the key `'main'`. When it reaches the bottom of this sub-tree, it decides how to continue by looking for a special IV named `'design'`. If this IV exists, it uses its value to decide which sub-tree to use next. When it reaches the end of these sub-trees, if there is an IV called `'design'` it again uses it to determine which sub-tree to use next. If there is no IV called `'design'`, then three tree ends. In other words, the possible values of the `'design'` IV should be names of sub-trees.

For example, let's make our experiment more complex by adding practice trials and two different session types. We'll add the practice trials by creating a new level called `'section'`, with the first section of each session proceeding to the practice trials, and the second into the experimental blocks.

We'll use the *`Experiment.from_yaml_file`* format:

```
design:
  main:
    - name: participant
      ivs:
        vision: [monocular, binocular]
      n: 20

    - name: session
      ivs:
        design: [first_session, second_session]
      ordering: Ordering

  first_session:
    - name: section
      ivs:
        design: [practice, first_experimental_section]
      ordering: Ordering

  second_session:
    - name: section
      ivs:
        design: [practice, second_experimental_section]
      ordering: Ordering

  practice:
    - name: trial
      ivs:
        difficulty: [easy]
        side: [left, right]
        display_time: [0.1, 1]
      practice: True

  first_experimental_section:
    - name: trial
      ivs:
        difficulty: [easy]
        side: [left, right]
        display_time: [0.1, 0.55, 1]
      n: 10
      practice: False

  second_experimental_section:
    - name: block
      ivs:
        difficulty: [easy, hard]

    - name: trial
      ivs:
        side: [left, right]
        display_time: [0.1, 0.55, 1]
      n: 5
      practice: False

filename: mixed_experiment.dat
```

Now we have a complex, heterogeneous experiment. Each participant will have two sessions; each session will start with four practice trials (a cross of two levels of the IV `'side'`, two levels of the IV `'display_time'`, and one level of the IV `'easy'`). The first session will include, after the practice section, sixty trials all with difficulty `'easy'`. The second session will include, after the practice session, two blocks in random order, the first with difficulty `'easy'` and the second `'hard'`, each with 30 trials. To make this happen we created four sub-trees in addition to the `'main` tree.

Note that we added the custom key `'practice'` to the `'trial'` level, to be able to more easily identify practice and experimental trials later (Alternatively, we could separate them later by looking for trials with `section==2` and ignoring trials with `section==1`). Also note that we use the *`Ordering`* method to produce a predictable order of the sub-trees. Otherwise, *`Shuffle`* is the default and we would get our sub-trees in a random order. Sometimes this is what we want, however. Because sub-trees are determined based on IV values, we can manipulate them in the same way as with other IVs, with ordering methods, design matrices, and even crossing them with other "normal" IVs.

It is not necessary to have the same level names for all possible routes down the tree. In this example, there are no blocks in the first session (or the practice section of the second session, for that matter). However, it is critical that all IVs get assigned a value in one place or another. In this example, the only place that the IV `'difficulty'` can take the value `'hard'` is at the `'block'` level of the second session. In other places on the design tree, we have to create an IV `'difficulty'` with only one level (`'easy'`) to ensure that we never generate a trial without assigning a value to the IV `'difficulty'`.

Manually modifying experiments

Another way to create complex experiment structures is to first construct a simple experiment, then manually modify it. For example, you can use the method *`ExperimentSection.append_child`* to add a child under any given section, or *`ExperimentSection.append_design_tree`* to add an entire sub-tree. See these methods' docstrings for details. Be sure to call *`Experiment.save`* after to make the changes permanent.

Design matrices

In all the examples so far, we've only specified possible IV values; we let experimentator handle the creation of conditions of them. Experimentator will use a full factorial cross, constructing a condition for every possible combination of IV values. Sometimes this isn't what we want, though. In a fractional factorial design, for example, only a subset of the possible combinations are used. We can specify these, and other, designs in experimentator using design matrices.

The support for design matrices in experimenator is designed to be compatible with the Python library pyDOE. This is a library that allows for easy creation of various common design matrices.

Design matrices can be specified during the creation of *`Design`* objects. This is the same place where IVs are specified when using the *Specification-based constructor methods*.

Each column of the design matrix is associated with one IV; a design matrix should have the same number of columns as the number of IVs in the design at that level. The order of IVs is important when using design matrices; because dictionaries in Python have no inherent order, `OrderedDict` should be used when defining IVs with design matrices, or alternatively IVs can be specified as a list of tuples (see the *IV docs*).

Each row of the design matrix is one condition, and the values of the matrix are interpreted in one of two ways:

- If the levels of an IV are passed as `None` rather than a list, then the IV is assumed to take arbitrary, continuous values. The values in the associated column of the design matrix are then interpreted at "face value".

- Otherwise, each value in the design matrix is interpreted as an index, determining which value to take from the list of possible IV values. Experimentator is smart about this and only cares about the relative value of these "indices". For example, if a design matrix column contains the values 0 and 1, they will be associated with the first and second IV values, respectively. Alternatively, if the column contains 1 and 2, then 1 will be associated with the first and 2 the second IV value.

A design matrix can also specify the order of conditions, by the order of its rows. For this reason, the default ordering method is *Ordering* when a design matrix is used. Change this to *Shuffle*, for example, if you instead want the rows of the design matrix to appear in a random order.

Here is an example of using a Box-Behnken design with pyDOE:

```
>>> import pyDOE
>>> from experimentator import Design
>>> design_matrix = pyDOE.bbdesign(3)
>>> print(design_matrix)
[[-1. -1.  0.]
 [ 1. -1.  0.]
 [-1.  1.  0.]
 [ 1.  1.  0.]
 [-1.  0. -1.]
 [ 1.  0. -1.]
 [-1.  0.  1.]
 [ 1.  0.  1.]
 [ 0. -1. -1.]
 [ 0.  1. -1.]
 [ 0. -1.  1.]
 [ 0.  1.  1.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
>>> trial_design = Design.from_dict(dict(
...     ivs=[('target_size', [10, 20, 30]),
...          ('target_speed', [5, 10, 20]),
...          ('target_position', None)],
...     design_matrix=design_matrix,
... ))
>>> # The following is just to demonstrate the conditions that are created.
>>> # These methods are usually called behind the scenes.
>>> trial_design.first_pass()
IndependentValue(name=(), values=())
>>> trial_design.get_order()
[{'target_position': 0.0, 'target_size': 10, 'target_speed': 5},
 {'target_position': 0.0, 'target_size': 30, 'target_speed': 5},
 {'target_position': 0.0, 'target_size': 10, 'target_speed': 20},
 {'target_position': 0.0, 'target_size': 30, 'target_speed': 20},
 {'target_position': -1.0, 'target_size': 10, 'target_speed': 10},
 {'target_position': -1.0, 'target_size': 30, 'target_speed': 10},
 {'target_position': 1.0, 'target_size': 10, 'target_speed': 10},
 {'target_position': 1.0, 'target_size': 30, 'target_speed': 10},
 {'target_position': -1.0, 'target_size': 20, 'target_speed': 5},
 {'target_position': -1.0, 'target_size': 20, 'target_speed': 20},
 {'target_position': 1.0, 'target_size': 20, 'target_speed': 5},
 {'target_position': 1.0, 'target_size': 20, 'target_speed': 20},
 {'target_position': 0.0, 'target_size': 20, 'target_speed': 10},
 {'target_position': 0.0, 'target_size': 20, 'target_speed': 10},
 {'target_position': 0.0, 'target_size': 20, 'target_speed': 10}]
```

Design.get_order (usually called behind the scenes) gives us a list of conditions, each a dictionary. We can see here the correspondence between the design matrix and the conditions. Because we used `None` with `'target_position'`, its values are taken directly from the matrix. For the other IVs, the values are taken from the list of possible values that we defined them with.

Callbacks

Up to this point, we've explained how to create an experiment of arbitrary complexity. But presumably you actually something to happen when you run a trial. This is accomplished with *callbacks*. In general, a callback is a function that you supply that is automatically triggered at a certain time. There are two types of callbacks in experimentator, the *function callbacks* and *context-managers*. Both are set with `Experiment.add_callback`.

Note: Be sure to save your experiment to disk after setting a callback, using `Experiment.save`, to make the changes permanent.

Note: Experimentator does not store the callbacks with your `Experiment`, but rather every time you load your experiment, the callbacks are re-imported. Experimentator looks for a Python file with the same name as the functions were originally defined in. As a result, the data file exported by `Experiment.save` is not sufficient when you want to move an experiment between computers. You will also need to move the Python file(s) in which any callbacks are defined.

`Experiment.add_callback` also takes optional keyword arguments `func_name` and `func_module` that you can set to tell experimentator where to look for the callback.

Function callbacks

The most basic callbacks are function callbacks. A function callback runs at the start of every section at its level. Most commonly, this is used at the trial level to set the "trial function"; on other words, the behavior of every trial.

Callbacks should take two positional arguments. It will be passed the current `Experiment` and `ExperimentSection` instances, respectively. Everything that the run callback might need to know can be taken from these arguments. Here are the most useful attributes:

- `ExperimentSection.data`: This is the `ChainMap` that contains the condition (IV values) for the currently running trial. It also includes the section numbers, for example `section.data['trial']` will get the current trial number.
- `Experiment.experiment_data`: This is a dictionary that you can use to store persistent data that every callback will have access to. By default, it is empty, but you can put data in here and it will always be available, even across sessions of the Python interpreter. This means that everything you put here must be picklable, so not everything will work.
- `Experiment.session_data`: This is where you can store data that is only persistent within the current session of the Python interpreter. Every time Python exits, this dictionary is emptied. This means you can store data here even if it is not picklable. This is the place to store external resources like multimedia data. You can reload these resources during a *context-manager callback*.

The callback should return a dictionary, mapping dependent variable (DV) names to values. The DV names are only used to label the columns in the final representation of the experiment's data, `Experiment.dataframe`.

Set function callbacks using the `Experiment.add_callback` method. You can also pass this method arbitrary positional and keyword arguments. Therefore, the full signature for a callback is `func(experiment, section, *args, **kwargs)`, where `func` (the callback itself), `*args`, and `**kwargs`, are arguments to `Experiment.add_callback`.

Context-managers

The second type of callback is the context manager. The name context manager is taken from the Python standard library, where they are referred to as With Statement Context Managers (the `with` statement is one way to *use* context managers, but it is not generally used to *create* them). Fundamentally, a context manager specifies behavior that should occur *before* something, and behavior that should occur *after*. In experimentator, the idea is that you will use context managers to define behavior that occurs before, between, and after sections of the experiment. One may want to open external resources (e.g., a sound file) at the beginning of each session, and close them afterward, for example. Another common use case would be to offer a break between blocks.

The most verbose way to create a context manager is to make a class that contains the magic methods `__enter__` and `__exit__` with "before" and "after" behavior, respectively. See Context Manager Types.

A much more convenient way is to use the `contextlib.contextmanager` decorator in the standard library. See the documentation for details, but it works like this: first you code the "before" behavior, then the keyword `yield`, then the "after" behavior. Here is an example context manager that offers a break between blocks:

```
from contextlib import contextmanager

@contextmanager
def offer_break(experiment, section):
    # Don't need to offer a break before the first block.
    if section.data['block'] > 1:
        input('Take a break if you would like.\nPress ENTER when you are ready to continue.')

    yield
    print('Block {} completed.'.format(section.data['block']))
```

As you see, the signature of a context manager is the same as the signature of a function callback. All the same data in the *`Experiment`* and *`ExperimentSection`* objects are also available to context managers.

Note: In the above example, we could make the `offer_break` function work on any level of the experiment. Every *`ExperimentSection`* stores its level name in the attribute `level`. If we replace `section.data['block']` with `section.data[section.level]` (we'd want to change the `print` message as well), then we could use `offer_break` at multiple levels.

Context-manager callbacks have the same signature as regular function callbacks, and are the added the same way. The only exception is to pass the keyword argument `is_context=True` to *`Experiment.add_callback`*.

With both types of callback, pass the level name to *`Experiment.add_callback`*. Continuing the previous example:

```
experiment.add_callback('block', offer_break, is_context=True)
```

If you are using the context manager to close resources, it may be a good idea to use a try-finally block (see Defining Clean-up Actions) to ensure that the resource is still closed in the case of an exception occurring. Here is an example that loads audio using the library pyglet:

```
from contextlib import contextmanager
import pyglet

@contextmanager
def load_audio(experiment, section):
    player = pyglet.media.Player()
    source = pyglet.media.load('background_music.mp3')

    # Make the Player available to other callbacks by saving it in session_data.
    experiment.session_data['player'] = player

    try:
        # Run the section.
        yield

    finally:
        # This block will run even if an error occurs during the try block.
        # If no error occurs, it will run after the section ends.
        player.delete()
```

Note: This example is just for illustration. Pyglet is actually smart enough to delete `player` for you when the Python interpreter exits.

An alternative to manually editing `Experiment.session_data` is to put objects after the `yield` keyword. Anything yielded by a context manager is stored in `experiment.session_data[level_name]` for the duration of the session. In the above example, if we have `yield player`, then we can access `player` from other callbacks as `experiment.session_data['session']` (assuming `load_audio` is set as the context manager of the level `'session'`).

5.1.3 Command-line interface

You've generated your experiment, now what? A major feature of experimentator is that it automatically turns your experiment into a command-line program, via the command `exp`. The general format for running experimentator commands is:

```
exp COMMAND <exp-file> OPTIONS
```

The available commands are *run*, *resume*, and *export*. Additionally, `exp --help` (or `-h`) will show the usage information, and `exp --version` will print experimentator's version number.

Commands

Here is the abridged usage message:

```
  exp run [options] <exp-file> (--next=<level>  [--not-finished] | (<level> <n>)... [--from=<n>])
  exp resume [options] <exp-file> (<level> | (<level> <n>)...)
  exp export <exp-file> <data-file> [ --no-index-label --delim=<sep> --skip=<columns> --float=<format
  exp -h | --help
  exp --version
```

Don't fear, an in-depth explanation follows.

run

The central command is `run`. There are a few ways to call it.

run –next To run the first section at `<level>` that hasn't been started:

```
exp run <exp-file> --next <level>
```

For example, to run the next participant from `example.exp`:

```
exp run example.exp --next participant
```

To run the next participant that hasn't been *finished* (as opposed to the next that hasn't *started*, the default):

```
exp run example.exp --next participant --not-finished
```

run <level> <n> To run a specific section:

```
exp run <exp-file> (<level> <n>)...
```

The elipsis means that the previous element (the pair `<level> <n>`) can be repeated any number of times. For example, to run the second session of the third participant:

```
exp run example.exp participant 3 session 2
```

run –from In either version of `run`, you can add the `--from <n>` option to start at a specific section. For example, to run the second session of the third participant, starting at the second block:

```
exp run example.exp participant 3 session 2 --from 2
```

`<n>` can also be a comma-separated list of integers. To start at the fourth trial of the second block:

```
exp run example.exp participant 3 session 2 --from 2,4
```

`--from=<n>` works the same as the `from_section` parameter to `run_experiment_section`; see documentation for that method for details.

run options Here is the full set of options for the `run` command:

-d, --debug	Set logging level to DEBUG.
-o <options>	Pass <options> to the experiment and save it as string in Experiment.session_data['options'].
--demo	Don't save data.
--not-finished	Run the first <level> that hasn't finished (rather than first that hasn't started).
--skip-parents	Don't enter context of parent levels.
--from=<n>	Start running at child number <n> of the specified section. <n> can also be a comma-separated list of ints; see run_experiment_section for details (specifically, –from=<n> works like the parameter from_section).

resume

`resume` is similar to `run`. There is nothing `resume` can do that `run` cannot, but `resume` makes resuming interrupted session easier. There are two ways to call it.

If you only pass a level, experimentator will try to resume the first section at that level that has been started but not finished. The syntax is:

```
exp resume <exp-file> <level>
```

For example, to resume the first block that has been started but not finished:

```
exp resume example.expblock
```

One can also use specific section numbers with `resume`:

```
exp resume <exp-file> (<level> <n>)...
```

The specified section must have been started but not finished. For example:

```
exp resume example.expparticipant 3 session 2
```

The difference between the above example and using `run` is that with `resume`, experimentator will automatically start at the appropriate place; with `run`, experimentator will start at the beginning of the section (unless an explicit starting point is passed with `--from`).

`resume` takes the same `options` as `run`.

export

`export` generates a text file with the experiment's data. Its basic syntax is:

```
exp export <exp-file> <data-file>
```

This one should be straightforward, but here is an example anyway:

```
exp export example.expexample.csv
```

Its associated options:

--no-index-label	Don't put column labels on index columns (e.g. participant, trial), for easier importing into R.
--delim=<sep>	Field delimiter [default: ,].
--skip=<columns>	Comma-separated list of columns to skip.
--float=<format>	Format string for floating point numbers.
--nan=<rep>	Missing data representation.

See `pandas.DataFrame.to_csv` for details on these options.

Note: If your experiment has any complex data structures (e.g., a timeseries for every trial), it is not recommended to use the `export` command, as this will create an unparseable mess. Instead, access your data programmatically through the `Experiment.dataframe` attribute.

5.1.4 API reference

Experiment

class `experimentator.`**`Experiment`**(*tree*, *data=None*, *has_started=False*, *has_finished=False*, *_children=None*, *filename=None*, *callback_by_level=None*, *callback_type_by_level=None*, *session_data=None*, *experiment_data=None*, *_callback_info=None*)

Bases: *`experimentator.section.ExperimentSection`*

An *`ExperimentSection`* subclass that represents the largest 'section' of the experiment; that is, the entire experiment. Functionality added on top of *`ExperimentSection`* includes various constructors, saving to disk, and management of *callbacks*.

To create a new experiment, rather than instantiating directly it is recommended to use one of the constructor methods:

- *`Experiment.new`*
- *`Experiment.from_dict`*
- *`Experiment.from_yaml_file`*
- *`Experiment.within_subjects`*
- *`Experiment.blocked`*
- *`Experiment.basic`*

Attributes

tree	(*`DesignTree`*) The *`DesignTree`* instance defining the experiment's hierarchy.
filename	(str) The file location where the *`Experiment`* will be pickled.
callback_by_level	(dict) A dictionary, mapping level names to functions or With Statement Context Managers (e.g., generator functions decorated with `contextlib.contextmanager`). Defines behavior to run at each section (for functions) or before and/or after each section (for context managers) at the associated level.
callback_type_by_level	(dict) A dictionary mapping level names to either the string `'context'` or `'function'`. This keeps track of which callbacks in `Experiment.callback_by_level` are context managers.
session_data	(dict) A dictionary where temporary data can be stored, persistent only within one session of the Python interpreter. This is a good place to store external resources that aren't picklable; external resources, for example, can be loaded in a context-manager callback and stored here. In addition, anything returned by the `__exit__` method of a context-manager callback will be stored here, with the callback's level name as the key. This dictionary is emptied before saving the *`Experiment`* to disk.
experiment_data	(dict) A dictionary where data can be stored that is persistent across Python sessions. Everything stored here must be picklable.

`add_callback`(*level*, *callback*, **args*, *is_context=False*, *func_module=None*, *func_name=None*, ***kwargs*)

Add a callback to run at a certain level.

A callback can be either a regular function, or a context-manager. The latter is useful for defining code to run at the start and end of every section at the level. For example, a block context manager could specify behavior that occurs before every trial in the block, and behavior that occurs after every trial in the block. See `contextlib` for various ways to create context managers, and experimentator's *context-manager docs* for more details.

Any value returned by the `__enter__` method of a context manager will be stored in `Experiment.session_data` under the key *level*.

If the callback is not a context manager, it should return a dictionary (or nothing), which is automatically passed to *`ExperimentSection.add_data`*. In theory, it should map dependent-variable names to results. See the *callback docs* for more details.

Parameters **level** : str

Which level of the hierarchy to manage.

callback : function or context-manager

The callback should have the signature `callback(experiment, section, *args, **kwargs)` where *experiment* and *section* are the current *`Experiment`* and *`ExperimentSection`* instances, respectively, and *args* and *kwargs* are arbitrary arguments passed to this method.

***args**

Any arbitrary positional arguments to be passed to *callback*.

func_module : str, optional

func_name : str, optional

These two arguments specify where the given function should be imported from in future Python sessions (i.e., `from <func_module> import <func_name>`). Usually, this is figured out automatically by introspection; these arguments are provided for the rare situation where introspection fails.

****kwargs**

Any arbitrary keyword arguments to be passed to *callback*.

`add_data`(*data*)

Update the `ExperimentSection.data` `ChainMap`. This data will apply to this section and all child sections. This can be used, for example, to manually record a participant's age.

Parameters **data** : dict

Elements to be added to `ExperimentSection.data`.

`all_subsections`(***section_numbers*)

Find all subsections in the experiment matching the given section numbers.

Yields specified *`ExperimentSection`* instances. The yielded sections will be at the lowest level given in *section_numbers*. If levels not in *section_numbers* are encountered before reaching its lowest level, all sections will be descended into.

Parameters ****section_numbers**

Keyword arguments describing what subsections to find. Keys are level names, values are ints or sequences of ints.

See also:

`experimentator.section.ExperimentSection.subsection`

Examples

Assuming the levels of the experiment saved in `'example.exp'` are `('participant', 'session', 'block', 'trial')`:

```
>>> from experimentator import Experiment
>>> exp = Experiment.load('example.exp')
```

Get the first session of each participant:

```
>>> all_first_sessions = list(exp.all_subsections(session=1))
```

Get the second trial of the first block in each session:

```
>>> trials = list(exp.all_subsections(block=1, trial=2))
```

Get the first three trials of each block in the first session of each participant:

```
>>> more_trials = list(exp.all_subsections(session=1, trial=[1, 2, 3]))
```

append_child(*data*, *tree=None*, *to_start=False*, *_renumber=True*)

Create a new *ExperimentSection* (and its descendants) and append it as a child of the current *ExperimentSection*.

Parameters **data** : dict

Data to be included in the new section's `ExperimentSection.data` `ChainMap`. Should include values of IVs at the section's level, for example.

tree : *DesignTree*, optional

If given, the section will be appended from the top level of *tree*. If not passed, the tree of the current section will be used. Note that this does not affect IV values; IV values must still be included in *data*.

to_start : bool, optional

If True, the new *ExperimentSection* will be appended to the beginning of the current section. If False (the default), it will be appended to the end.

Notes

After calling this method, the section numbers in the children's `ExperimentSection.data` attributes will be automatically replaced with the correct numbers.

append_design_tree(*tree*, *to_start=False*, *_renumber=True*)

Append all sections associated with the top level of a *DesignTree* (and therefore also create descendant sections) to the *ExperimentSection*.

Parameters **tree** : *DesignTree*

The tree to append.

to_start : bool, optional

If True, the sections will be inserted at the beginning of the section. If False (the default), they will be appended to the end.

Notes

After calling this method, the section numbers in the children's `ExperimentSection.data` attributes will be automatically replaced with the correct numbers.

as_graph()

Build a `networkx.DiGraph` out of the experiment structure, starting at this section. Nodes are sections and graphs are parent-child relations. Node data are non-duplicated entries in `ExperimentSection.data`.

Returns `networkx.DiGraph`

classmethod basic(*levels*, *ivs_by_level*, *design_matrices_by_level=None*, *ordering_by_level=None*, *filename=None*)

Construct a homogeneously-organized *`Experiment`*, with arbitrary levels but only one *`Design`* at each level, and the same structure throughout its hierarchy.

Parameters **levels** : sequence of str

Names of the levels of the experiment

ivs_by_level : dict

Dictionary specifying the IVs and their possible values at every level. The keys are be the level names, and the values are lists of the IVs at that level, specified in the form of tuples with the first element being the IV name and the second element a list of its possible values. Alternatively, the IVs at each level can be specified in a dictionary. See *IV docs* for more on specifying IVs.

design_matrices_by_level : dict, optional

Specify the design matrix for any levels. Keys are level names; values are design matrices. Any levels without a design matrix will be fully crossed. See *design matrix docs* for details.

ordering_by_level : dict, optional

Specify the ordering for each level. Keys are level names; values are instance objects from *`experimentator.order`*. For any levels without an order specified, *`Shuffle`* will be used.

filename : str, optional

File location to save the experiment.

Returns *`Experiment`*

classmethod blocked(*trial_ivs*, *n_participants*, *design_matrices=None*, *orderings=None*, *block_ivs=None*, *filename=None*)

Create a blocked within-subjects *`Experiment`*, in which all the IVs are at either the trial level or the block level.

Parameters **trial_ivs** : list or dict

A list of the IVs to define at the trial level, specified in the form of tuples with the first element being the IV name and the second element a list of its possible values. Alternatively, the IVs at each level can be specified in a dictionary. See the *IV docs* more on specifying IVs.

n_participants : int

Number of participants to initialize. If a *`NonAtomicOrdering`* is used, this is the number of participants per order.

design_matrices : dict, optional

Design matrices for the experiment. Keys are `'trial'` and `'block'`; values are the respective design matrices (if any). If not specified, IVs will be fully crossed. See the *design matrix docs* for details.

orderings : dict, optional

Dictionary with keys of `'trial'` and `'block'`. Each value should be an instance of the class `Ordering` or one of its subclasses, specifying how the trials will be ordered If not specified, `Shuffle` will be used.

block_ivs : list or dict, optional

IVs to define at the block level. See *IV docs* for more on specifying IVs.

filename : str, optional

File location to save the experiment.

Returns `Experiment`

Notes

For blocks to have any effect, you should either define at least one IV at the block level or use the ordering `Ordering(n)` to create `n` blocks for every participant.

`breadth_first_search`(*key*)
Breadth-first search starting from here. Returns the entire search path.

Parameters **key** : func

Function that returns True or False when passed an `ExperimentSection`.

Returns list of `ExperimentSection`

`depth_first_search`(*key*, *path_key=None*, *_path=None*)
Depth-first search starting from here. Returns the entire search path.

Parameters **key** : func

Function that returns True or False when passed an `ExperimentSection`.

path_key : func, optional

Function that returns True or False when passed an `ExperimentSection`. If given, the search will proceed only via sections for which *path_key* returns True.

Returns list of `ExperimentSection`

`export_data`(*filename*, *skip_columns=None*, ***kwargs*)
Export `Experiment.dataframe` in `.csv` format.

Parameters **filename** : str

A file location where the data should be saved.

skip_columns : list of str, optional

Columns to skip.

****kwargs**

Arbitrary keyword arguments to pass to `pandas.DataFrame.to_csv`.

Notes

This method is not recommended for experiments with compound data types, for example an experiment which stores a time series for every trial. In those cases it is recommended to write a custom script that parses the `Experiment.dataframe` attribute as desired, or use the *skip_columns* option to skip any compound columns.

find_first_not_run(*at_level*, *by_started=True*)
Search the experimental hierarchy, and return the first descendant *ExperimentSection* at *at_level* that has not yet been run.

Parameters **at_level** : str

Which level to search.

by_started : bool, optional

If True (default), returns the first section that has not been started. Otherwise, finds the first section that has not finished.

Returns *ExperimentSection*

find_first_partially_run(*at_level*)
Search the experimental hierarchy, and return the first descendant *ExperimentSection* at *at_level* that has been started but not finished.

Parameters **at_level** : str

Which level to search.

Returns *ExperimentSection*

classmethod from_dict(*spec*)
Construct an *Experiment* based on a dictionary specification.

Parameters **spec** : dict

spec should have, at minimum, a key named `'design'`. The value of this key specifies the *DesignTree*. See *DesignTree.from_spec* for details. The value of the key `'filename'` or `'file'`, if one exists,is saved in `Experiment.filename`. All other fields are saved in `Experiment.experiment_data`.

Returns *Experiment*

See also:

experimentator.Experiment.from_yaml_file

classmethod from_yaml_file(*filename*)
Construct an *Experiment* based on specification in a YAML file. Requires PyYAML.

Parameters **filename** : str

YAML file location. The YAML should specify a dictionary matching the specification of *Experiment.from_dict*.

Returns *Experiment*

get_next_tree()
Get a tree to use for creating child *ExperimentSection* instances.

Returns *DesignTree*

static load(*filename*)
Load an experiment from disk.

Parameters **filename** : str

Path to a file generated by *Experiment.save*.

Returns *Experiment*

classmethod new (*tree*, *filename=None*)
Make a new *Experiment*.

Parameters **tree** : *DesignTree*

A *DesignTree* instance defining the experiment hierarchy.

filename : str, optional

A file location where the *Experiment* will be saved.

parent (*section*)
Find the parent of a section.

Parameters **section** : *ExperimentSection*

The section to find the parent of.

Returns *ExperimentSection*

parents (*section*)
Find all parents of a section, in top-to-bottom order.

Parameters **section** : *ExperimentSection*

The section to find the parents of.

Returns list of *ExperimentSection*

resume_section (*section*, ***kwargs*)
Rerun a section that has been started but not finished, starting where running last left off.

Parameters **section** : *ExperimentSection*

The section to resume.

****kwargs**

Keyword arguments to pass to *Experiment.run_section*.

Notes

The wrapper function `run_experiment_section` should be used instead of this method, if possible.

run_section (*section*, *demo=False*, *parent_callbacks=True*, *from_section=None*)
Run a section and all its descendant sections. Saves the results in the `data` attribute of each lowest-level *ExperimentSection*.

Parameters **section** : *ExperimentSection*

The section to be run.

demo : bool, optional

Data will only be saved if *demo* is False (the default).

parent_callbacks : bool, optional

If True (the default), all parent callbacks will be called.

from_section : int or list of int, optional

Which section to start running from. If a list is passed, it specifies where to start running on multiple levels. For example, assuming the experiment hierarchy is `('participant', 'session', 'block', 'trial')`, this would start from the fifth trial of the second block (of the first participant's second session):

```
>>> exp = Experiment.load('example.exp')
>>> exp.run_section(exp.subsection(participant=1, session=2), from_section=[2, 5])
```

Notes

The wrapper function `run_experiment_section` should be used instead of this method, if possible.

save (*filename=None*)
Save the *Experiment* to disk.

Parameters **filename** : str, optional

If specified, overrides `Experiment.filename`.

subsection (***section_numbers*)
Find a single, descendant *ExperimentSection* based on section numbers.

Parameters ****section_numbers**

Keyword arguments describing which subsection to find Must include every level higher than the desired section.

Returns *ExperimentSection*

See also:

experimentator.section.ExperimentSection.all_subsections

Examples

Assuming the levels of the experiment saved in `'example.exp'` are `('participant', 'session', 'block', 'trial')`, this will return the third block of the second participant's first session:

```
>>> from experimentator import Experiment
>>> exp = Experiment.load('example.exp')
>>> some_block = exp.subsection(participant=2, session=1, block=3)
```

walk ()
Walk the tree depth-first, starting from here. Yields this section and every descendant section.

classmethod within_subjects (*ivs, n_participants, design_matrix=None, ordering=None, filename=None*)
Create a within-subjects *Experiment*, with all the IVs at the `'trial'` level.

Parameters **ivs** : list or dict

A list of the experiment's IVs, specified in the form of tuples with the first element being the IV name and the second element a list of its possible values. Alternatively, the IVs at each level can be specified in a dictionary. See the *IV docs* more on specifying IVs.

n_participants : int

Number of participants to initialize.

design_matrix : array-like, optional

Design matrix for the experiment. If not specified, IVs will be fully crossed. See the *design matrix docs* for more details.

ordering : *Ordering*, optional

An instance of the class `Ordering` or one of its subclasses, specifying how the trials will be ordered. If not specified, `Shuffle` will be used.

filename : str, optional

File location to save the experiment.

Returns `Experiment`

Helper functions

`experimentator.`**`run_experiment_section`**(*experiment*, *section_obj=None*, *demo=False*, *resume=False*, *parent_callbacks=True*, *from_section=1*, *session_options=''*, ***section_numbers*)

Run an experiment from a file or an `Experiment` instance, and save it. If an exception is encountered, the `Experiment` will be backed up and saved.

Parameters **experiment** : str or `Experiment`

File location where an `Experiment` instance is pickled, or an `Experiment` instance.

demo : bool, optional

If True, data will not be saved and sections will not be marked as run.

resume: bool, optional

If True, the specified section will be resumed (started automatically where it left off).

parent_callbacks : bool, optional

If True (the default), all parent callbacks will be called.

section_obj : `ExperimentSection`, optional

The section of the experiment to run. Alternatively, the section can be specified using ***section_numbers*.

from_section : int or list of int, optional

Which section to start running from. If a list is passed, it specifies where to start running on multiple levels. See the example below.

session_options : str, optional

Pass an experiment-specific options string to be stored in `Experiment.session_data` under the key `'options'`.

****section_numbers**

Keyword arguments describing how to descend the experiment hierarchy to find the section to run. See the example below.

Examples

A simple example:

```
>>> exp = Experiment.load('example.exp')
>>> run_experiment_section(exp, exp.subsection(participant=1, session=2))
```

Equivalently:

```
>>> run_experiment_section(exp, participant=1, session=2)
```

To demonstrate *from_section*, assuming the experiment hierarchy is `('participant', 'session', 'block', 'trial')`, this would start from the second block:

```
>>> run_experiment_section(exp, participant=1, session=2, from_section=2)
```

To start from the fifth trial of the second block:

```
>>> run_experiment_section(exp, participant=1, session=2, from_section=[2, 5])
```

`experimentator.`**`export_experiment_data`**(*exp_filename*, *data_filename*, ***kwargs*)

Reads a pickled `Experiment` instance and saves its data in `.csv` format.

Parameters **exp_filename** : str

The file location where an `Experiment` instance is pickled.

data_filename : str

The file location where the data will be written.

skip_columns : list of str, optional

Data columns to skip.

****kwargs**

Arbitrary keyword arguments passed through to `pandas.DataFrame.to_csv`.

Notes

This shortcut function is not recommended for experiments with compound data types, for example an experiment which stores a time series for every trial. In such cases it is recommended to write a custom script that parses `Experiment.dataframe` as desired (or use the *skip_columns* option to ignore the compound data).

ExperimentSection

class `experimentator.section.`**`ExperimentSection`**(*tree*, *data=None*, *has_started=False*, *has_finished=False*, *_children=None*)

A section of the experiment, at any level of the hierarchy. Single trials and groups of trials (blocks, sessions, participants, etc.) are represented as `ExperimentSection` instances. A complete experiment consists of `ExperimentSection` instances arranged in a tree. The root element should be an `Experiment` (a subclass of `ExperimentSection`); the rest of the sections can be reached via its descendants (see below on the sequence protocol). A new `ExperimentSection` instance is automatically populated with `ExperimentSection` descendants according to the `DesignTree` passed to its constructor.

`ExperimentSection` implements Python's sequence protocol; its contents are `ExperimentSection` instances at the level below. In other words, children can be accessed using the `[index]` notation, as well as with slices (`[3:6]`) or iteration (`for section in experiment_section`). However, `ExperimentSection` breaks the Python convention of 0-based indexing, using 1-based indexing to match the convention in experimental science.

The direct constructor is used to create an arbitrary `ExperimentSection` (i.e., possibly reloading an in-progress section), whereas `ExperimentSection.new` creates a section that hasn't yet started.

Notes

Use 1-based indexing to refer to `ExperimentSection` children, both when when using indexing or slicing with an `ExperimentSection`, and when identifying sections in keyword arguments to methods such as `ExperimentSection.subsection`. This better corresponds to the language commonly used by scientists to identify participants, trials, etc.

Attributes

tree	(`DesignTree`)
data	(`ChainMap`)
description	(str) The name and number of the section (e.g., `'trial 3'`).
dataframe	(`DataFrame`) All data associated with the `ExperimentSection` and its descendants.
heteroge-neous_design_iv_name	(str) IV name determining which branch of the `DesignTree` to follow.
level	(str) The level of the hierarchy at which this section lives.
levels	(list of str) Level names below this section.
local_levels	(set) Level names of this section's children. Usually a single-element set.
is_bottom_level	(bool) If true, this is the lowest level of the hierarchy.
is_top_level	(bool) If true, this is the highest level of the hierarchy (likely an `Experiment`).
has_started: bool	Whether this section has started to be run.
has_finished	(bool) Whether this section has finished running.

`add_data`(*data*)

Update the `ExperimentSection.data` `ChainMap`. This data will apply to this section and all child sections. This can be used, for example, to manually record a participant's age.

Parameters **data** : dict

Elements to be added to `ExperimentSection.data`.

`all_subsections`(***section_numbers*)

Find all subsections in the experiment matching the given section numbers.

Yields specified `ExperimentSection` instances. The yielded sections will be at the lowest level given in *section_numbers*. If levels not in *section_numbers* are encountered before reaching its lowest level, all sections will be descended into.

Parameters ****section_numbers**

Keyword arguments describing what subsections to find. Keys are level names, values are ints or sequences of ints.

See also:

`experimentator.section.ExperimentSection.subsection`

Examples

Assuming the levels of the experiment saved in `'example.exp'` are `('participant', 'session', 'block', 'trial')`:

```
>>> from experimentator import Experiment
>>> exp = Experiment.load('example.exp')
```

Get the first session of each participant:

```
>>> all_first_sessions = list(exp.all_subsections(session=1))
```

Get the second trial of the first block in each session:

```
>>> trials = list(exp.all_subsections(block=1, trial=2))
```

Get the first three trials of each block in the first session of each participant:

```
>>> more_trials = list(exp.all_subsections(session=1, trial=[1, 2, 3]))
```

append_child(*data*, *tree=None*, *to_start=False*, *_renumber=True*)

Create a new *ExperimentSection* (and its descendants) and append it as a child of the current *ExperimentSection*.

Parameters **data** : dict

Data to be included in the new section's `ExperimentSection.data` `ChainMap`. Should include values of IVs at the section's level, for example.

tree : *DesignTree*, optional

If given, the section will be appended from the top level of *tree*. If not passed, the tree of the current section will be used. Note that this does not affect IV values; IV values must still be included in *data*.

to_start : bool, optional

If True, the new *ExperimentSection* will be appended to the beginning of the current section. If False (the default), it will be appended to the end.

Notes

After calling this method, the section numbers in the children's `ExperimentSection.data` attributes will be automatically replaced with the correct numbers.

append_design_tree(*tree*, *to_start=False*, *_renumber=True*)

Append all sections associated with the top level of a *DesignTree* (and therefore also create descendant sections) to the *ExperimentSection*.

Parameters **tree** : *DesignTree*

The tree to append.

to_start : bool, optional

If True, the sections will be inserted at the beginning of the section. If False (the default), they will be appended to the end.

Notes

After calling this method, the section numbers in the children's `ExperimentSection.data` attributes will be automatically replaced with the correct numbers.

as_graph()

Build a `networkx.DiGraph` out of the experiment structure, starting at this section. Nodes are sections and graphs are parent-child relations. Node data are non-duplicated entries in `ExperimentSection.data`.

Returns `networkx.DiGraph`

`breadth_first_search`(*key*)

Breadth-first search starting from here. Returns the entire search path.

Parameters **key** : func

Function that returns True or False when passed an *`ExperimentSection`*.

Returns list of *`ExperimentSection`*

`depth_first_search`(*key*, *path_key=None*, *_path=None*)

Depth-first search starting from here. Returns the entire search path.

Parameters **key** : func

Function that returns True or False when passed an *`ExperimentSection`*.

path_key : func, optional

Function that returns True or False when passed an *`ExperimentSection`*. If given, the search will proceed only via sections for which *path_key* returns True.

Returns list of *`ExperimentSection`*

`find_first_not_run`(*at_level*, *by_started=True*)

Search the experimental hierarchy, and return the first descendant *`ExperimentSection`* at *at_level* that has not yet been run.

Parameters **at_level** : str

Which level to search.

by_started : bool, optional

If True (default), returns the first section that has not been started. Otherwise, finds the first section that has not finished.

Returns *`ExperimentSection`*

`find_first_partially_run`(*at_level*)

Search the experimental hierarchy, and return the first descendant *`ExperimentSection`* at *at_level* that has been started but not finished.

Parameters **at_level** : str

Which level to search.

Returns *`ExperimentSection`*

`get_next_tree`()

Get a tree to use for creating child *`ExperimentSection`* instances.

Returns *`DesignTree`*

classmethod `new`(*tree*, *data=None*)

Create a new *`ExperimentSection`*.

Parameters **tree** : *`DesignTree`*

Describes the design of the experiment hierarchy.

data : `ChainMap`

All data to be associated with the *`ExperimentSection`*, including the values of independent variables, the section numbers indicating the section's location in the experiment, and any results associated with this section, arising from either the run callback of the `Experiment` or from the method `ExperimentSection.add_data`. *data* should be a `collections.ChainMap`, which behaves like a dictionary but has a hierarchical organization such that children can access values from the parent but not vice-versa.

`parent` (*section*)

Find the parent of a section.

Parameters **section** : `ExperimentSection`

The section to find the parent of.

Returns `ExperimentSection`

`parents` (*section*)

Find all parents of a section, in top-to-bottom order.

Parameters **section** : `ExperimentSection`

The section to find the parents of.

Returns list of `ExperimentSection`

`subsection` (***section_numbers*)

Find a single, descendant `ExperimentSection` based on section numbers.

Parameters ****section_numbers**

Keyword arguments describing which subsection to find Must include every level higher than the desired section.

Returns `ExperimentSection`

See also:

`experimentator.section.ExperimentSection.all_subsections`

Examples

Assuming the levels of the experiment saved in `'example.exp'` are `('participant', 'session', 'block', 'trial')`, this will return the third block of the second participant's first session:

```
>>> from experimentator import Experiment
>>> exp = Experiment.load('example.exp')
>>> some_block = exp.subsection(participant=2, session=1, block=3)
```

`walk` ()

Walk the tree depth-first, starting from here. Yields this section and every descendant section.

Design

class `experimentator.`**`Design`** (*ivs=None*, *design_matrix=None*, *ordering=None*, *extra_data=None*)

`Design` instances specify the experimental design at one level of the experimental hierarchy. They guide the creation of `ExperimentSection` instances by parsing design matrices or crossing independent variables (IVs).

Parameters **ivs** : dict or list of tuple, optional

Independent variables can be specified as a dictionary mapping names to possible values, or as a list of `(name, values)` tuples. If an IV takes continuous values, use `None` for its levels. This only works when specifying values using *design_matrix*. See the *IV docs* for more information.

design_matrix : array-like, optional

A `numpy array` (or convertible, e.g. a list-of-lists) representing a design matrix specifying how IV values should be grouped to form conditions. When no *design_matrix* is passed, IVs are fully crossed. See the *design matrix docs* for more details. Note that a design matrix may also specify the order of the conditions. For this reason, the default *ordering* changes from `Shuffle` to `Ordering`, preserving the order of the conditions.

ordering : `Ordering`, optional

An instance of `Ordering` or one of its subclasses defining the behavior for duplicating and ordering the conditions of the `Design`. The default is `Shuffle` unless a *design_matrix* is passed.

extra_data : dict, optional

Items from this dictionary will be included in the `data` attribute of any `ExperimentSection` instances created with this `Design`.

See also:

`experimentator.order`, `experimentator.DesignTree`

Examples

```
>>> from experimentator.order import Shuffle
>>> design = Design(ivs={'side': ['left', 'right'], 'difficulty': ['easy', 'hard']}, ordering=Sh
>>> design.first_pass()
IndependentVariable(name=(), values=())
>>> design.get_order()
[{'difficulty': 'easy', 'side': 'left'},
 {'difficulty': 'hard', 'side': 'left'},
 {'difficulty': 'easy', 'side': 'left'},
 {'difficulty': 'hard', 'side': 'right'},
 {'difficulty': 'easy', 'side': 'right'},
 {'difficulty': 'easy', 'side': 'right'},
 {'difficulty': 'hard', 'side': 'left'},
 {'difficulty': 'hard', 'side': 'right'}]
```

Attributes

iv_names	(list of str)
iv_values	(list of tuple)
design_matrix	(array-like)
extra_data	(dict)
ordering	(`Ordering`)
heterogeneous_design_iv_name	(str) The IV name that triggers a heterogeneous (i.e., branching) tree structure when it is encountered. `'design'` by default.
is_heterogeneous	(bool) True if this `Design` is the lowest level before the tree structure diverges.
branches	(dict) The IV values corresponding to named heterogeneous branches in the tree structure following this `Design`.

first_pass()

Initialize design.

Initializes the design by parsing the design matrix or crossing the IVs If a *NonAtomicOrdering* is used, an additional IV will be returned which should be incorporated into the design one level up in the experimental hierarchy. For this reason, the *first_pass* methods in a hierarchy of *Design* instances should be called in reverse order, from bottom up. Use a *DesignTree* to ensure this occurs properly.

Returns iv_name : str or tuple

The name of the IV, for *non-atomic orderings*. Otherwise, an empty tuple.

iv_values : tuple

The possible values of the IV. Empty for atomic orderings.

classmethod from_dict(*spec*)

Construct a *Design* instance from a specification based on dictionaries (e.g., parsed from a YAML file).

Parameters spec : dict

A dictionary containing some of the following keys (all optional): `'name'`, the name of the level; `'ivs'`, `'design_matrix'`, `'extra_data'`, keyword arguments to the *Design* constructor; `'order'` or `'ordering'`, a string, dictionary, or list determining the ordering method; and `'n'` or `'number'`, the `number` argument to the specified ordering. A dictionary containing any fields not otherwise used is passed to the *Design* constructor as the `extra_data` argument. See the *description in the docs* for more information.

Returns name : str

Only returned if *spec* contains a field `'name'`.

design : *Design*

See also:

experimentator.DesignTree.from_spec

Examples

```
>>> design_spec = {
...'name': 'block',
...'ivs': {'speed': [1, 2, 3], 'size': [15, 30]},
...'ordering': 'Shuffle',
...'n': 3}
>>> Design.from_dict(design_spec)
Level(name='block', design=Design(ivs=[('speed', [1, 2, 3]), ('size', [15, 30])], design_mat
```

static full_cross(*iv_names*, *iv_values*)

Perform a full factorial cross of the independent variables. Yields dictionaries, each describing one condition, a mapping from IV names to IV values. One dictionary is yielded for every possible combination of IV values.

Parameters iv_names : list of str

Names of IVs.

iv_values : list of list

Each element defines the possible values of an IV. Must be the same length as *iv_names*. Its elements must be hashable.

get_order(*data=None*)

Order the conditions.

Returns list of dict

A list of dictionaries, each specifying a condition (a mapping from IV names to values).

update(*names*, *values*)

Add additional independent variables to the *Design*. This will have no effect after *Design.first_pass* has been called.

Parameters **names** : list of str

Names of IVs to add.

values : list of list

For each IV, a list of possible values.

DesignTree

class experimentator.**DesignTree**(*levels_and_designs=None*, *other_designs=None*, *branches=None*)

A container for *Design* instances, describing the entire hierarchy of a basic *Experiment*. *DesignTree* instances are iterators; calling `next` on one will return another *DesignTree* with the top level removed. In this way, the entire experimental hierarchy can be created by recursively calling `next`.

Use *DesignTree.new* to create a new tree, the generic constructor is for instantiating trees whose attributes have already been processed (i.e., reloading already-created trees).

Notes

Calling `next` on the last level of a heterogeneous *DesignTree* will return a dictionary of named *DesignTree* instances (rather than a single *DesignTree* instance). The keys are the possible values of the IV `'design'` and the values are the corresponding *DesignTree* instances.

Attributes

levels_and_designs	(list of tuple)
other_designs	(dict)
branches	(dict) Only those items from *other_designs* that follow directly from this tree.

add_base_level()

Adds a section to the top of the tree called `'_base'`. This makes the *DesignTree* suitable for constructing an *Experiment*.

Notes

The *Experiment* constructor calls this automatically, and this shouldn't be called when appending a tree to an existing *Experiment*, so there is no use case for manually calling this method.

static first_pass(*levels_and_designs*)

Make a first pass of all designs in a *DesignTree*, from bottom to top. This calls *Design.first_pass* on every *Design* instance in the tree in the proper order, updating designs when a new IV is returned. This is necessary for *non-atomic orderings* because they modify the parent *Design*.

classmethod **`from_spec`**(*spec*)

Constructs a *`DesignTree`* instance from a specification (e.g., parsed from a YAML file).

spec [dict or list of dict] The *`DesignTree`* specification. A dictionary with keys as tree names and values as lists of dictionaries. Each sub-dictionary should specify a *`Design`* according to *`Design.from_dict`*. The main tree should be named `'main'`. Other names are used for generating heterogeneous trees (see *`DesignTree`* docs). A homogeneous tree can be specified as a dictionary with only a single key `'main'`, or directly as a list of dictionaries

Returns *`DesignTree`*

classmethod **`new`**(*levels_and_designs*, ***other_designs*)

Create a new *`DesignTree`*.

Parameters **levels_and_designs** : `OrderedDict` or list of tuple

This input defines the structure of the tree, and is either an `OrderedDict` or a list of 2-tuples. Keys (or first element of each tuple) are level names. Values (or second element of each tuple) are design specifications, in the form of either a *`Design`* instance, or a list of *`Design`* instances to occur in sequence.

****other_designs**

Named design trees, can be other *`DesignTree`* instances or suitable *levels_and_designs* inputs (i.e., `OrderedDict` or list of tuples). These designs allow for heterogeneous design structures (i.e. not every section at the same level has the same *`Design`*). To make a heterogeneous *`DesignTree`*, use an IV named `'design'` at the level where the heterogeneity should occur. Values of this IV should be strings, each corresponding to the name of a *`DesignTree`* from‘ other_designs‘. The value of the IV `'design'` at each section determines which *`DesignTree`* is used for children of that section.

experimentator.order

This module contains the class *`Ordering`* and its descendants. These classes handle how unique conditions at a particular experimental level are ordered and duplicated. *`Ordering`* instances should be passed directly to the *`Design`* constructor; there is no reason to otherwise interact with them in normal use.

Of special note are non-atomic orderings: the class *`NonAtomicOrdering`* and its descendants. ‘’Non-atomic” here means that the orderings between sections are not independent. A *`Shuffle`* ordering is atomic; the order in one section is independent of the order in another. However, if one wants to ensure, for example, that possible block orders are evenly distributed among participants (a *`counterbalanced design`*), the block orders within each participants are not independent. Each participant can decide its order of blocks only in the context of the other participants' block orders. This means that the *parent* section must handle orderings (in the example of counterbalanced blocks, the `Experiment.base_section`–the experiment itself, essentially–must tell each participant what block order to use).

class `experimentator.order.`**`Ordering`**(*number=1*)

Bases: `object`

The base ordering class. It will keep conditions in the order they are defined by the *`Design`* instance (either the order of rows in the design matrix, or a full factorial cross–the output of calling `itertools.product` on the IV levels). Remember not to rely on the order of dictionary items. Therefore, if a specific order is desired, it is recommended to use a design matrix or an `OrderedDict` to define the IVs. See the *IV docs* for more information.

Parameters **number** : int, optional

The number of times each unique condition should appear. The default is 1. If `number > 1`, the entire order will be cycled (as opposed to repeating each condition within the order).

`first_pass`(*conditions*)

Handle operations that should only be performed once, initializing the object before ordering conditions. For *`Ordering`*, the only operation is duplication of the list of conditions (if `Ordering.number` > 1). This methods should not be called manually.

Parameters **conditions** : sequence of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

Returns **iv_name** : str or tuple

The name of the IV, for non-atomic orderings. Otherwise, an empty tuple.

iv_values : tuple

The possible values of the IV. Empty for atomic orderings.

`get_order`(*data=None*)

Get an order of conditions. For *`Ordering`*, always returns the same order.

Parameters **data** : dict, optional

A dictionary describing the data of the parent section. Unused for atomic orderings.

Returns list of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

static **`possible_orders`**(*conditions*, *unique=True*)

Yield all possible orders of the conditions. Each order is a list of dictionaries, with each dictionary representing a condition.

Parameters **conditions** : sequence of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

unique : bool, optional

If true (the default), will only return unique orders. If false, some identical orders will be generated if *conditions* contains identical elements.

class `experimentator.order.`**`Shuffle`**(*number=1*, *avoid_repeats=False*)

Bases: *`experimentator.order.Ordering`*

This ordering randomly shuffles the conditions.

Parameters **number** : int, optional

Number of times each condition should appear (default=1). Conditions are duplicated *before* shuffling.

avoid_repeats : bool, optional

If True (default is False), no identical conditions will appear back-to-back.

`get_order`(*data=None*)

Get an order of conditions. For *`Shuffle`*, returns the conditions in a random order.

Parameters **data** : dict, optional

A dictionary describing the data of the parent section. Unused for atomic orderings.

Returns list of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

class `experimentator.order.`**`NonAtomicOrdering`**(*number=1*)

Bases: *`experimentator.order.Ordering`*

This is a base class for non-atomic orderings, and is not meant to be directly instantiated. Non-atomic orderings work by creating a new independent variable one level up. The IV name will start with an underscore, a convention to avoid name clashes.

`get_order`(*data=None*)

Get an order of conditions.

Parameters **data** : dict, optional

A dictionary describing the data of the parent section.

Returns list of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

`iv`

The IV associated with the non-atomic ordering. It will be added to the design one level up in the experiment hierarchy.

Returns **iv_name** : str or tuple

The name of the IV, for non-atomic orderings. Otherwise, an empty tuple.

iv_values : seq

The possible values of the IV. Empty for atomic orderings.

class `experimentator.order.`**`CompleteCounterbalance`**(*number=1*)

Bases: *`experimentator.order.NonAtomicOrdering`*

In a complete counterbalance design, every unique ordering of the conditions appears the same numbers of times.

Parameters **number** : int, optional

The number of times each condition should be duplicated. Note that conditions are duplicated *before* determining the possible orderings.

Notes

The number of possible orderings can get very large very quickly. Therefore, a complete counterbalance is not recommended for more than 3 conditions. The number of unique orderings can be determined by `factorial(number * k) // number**k`, where *k* is the number of conditions (assuming all conditions are unique). For example, with 5 conditions there are 120 possible orders; with 3 conditions and `number==2`, there are 90 unique orders.

`first_pass`(*conditions*)

Handle operations that should only be performed once, initializing the object before ordering conditions. For *`CompleteCounterbalance`*, all possible orders are determined. This method should not be called manually.

Parameters **conditions** : sequence of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

Returns **iv_name** : str or tuple

The name of the IV to be created one level up, `'counterbalance_order'`.

iv_values : tuple

Values of the IV to be created one level up, integers each associated with an order of the conditions.

class `experimentator.order.`**`Sorted`**(*number=1*, *order='both'*)

Bases: *`experimentator.order.NonAtomicOrdering`*

Sorts the conditions based on the value of the IV defined at its level. This ordering can be non-atomic (if *order* `== 'both'`), creating an IV with levels of `'ascending'` and `'descending'`. If *order* is `'ascending'` or `'descending'`, the ordering will be atomic (each section will be ordered the same way).

Parameters **order** : {'both', 'ascending', 'descending'}, optional

The order to sort the sections. If `'both'` (the default), half the sections will be created in ascending order, and half in descending order, depending on the value of the new IV `'sorted_order'`.

number : int, optional

The number of times each condition should appear.

Notes

To avoid ambiguity, *`Sorted`* can only be used at levels containing only one IV.

`first_pass`(*conditions*)

Handle operations that should only be performed once, initializing the object before ordering conditions. For *`Sorted`*, the conditions are sorted on IV values.

Parameters **conditions** : sequence of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

Returns **iv_name** : str or tuple

If *order* is `'both'`, the name of the IV to be created one level up, `'sorted_order'`. Otherwise, an empty tuple (denoting that no IV will be created).

iv_values : tuple

If *order* is `'both'`, values of the IV to be created one level up, integers each associated with an order of the conditions. Otherwise, empty tuple.

`get_order`(*data=None*)

Get an order of conditions.

Parameters **data** : dict, optional

A dictionary describing the data of the parent section.

Returns list of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

class `experimentator.order.`**`LatinSquare`**(*number=1*, *balanced=True*, *uniform=False*)

Bases: *`experimentator.order.NonAtomicOrdering`*

Orders the conditions by constructing an NxN Latin square, where N is the number of unique conditions. A Latin square is a matrix with each element appearing exactly once in each row and column. Each row represents a different potential ordering of the conditions. This allows for balanced counterbalancing, in designs too large to accommodate a complete counterbalance.

Parameters **number** : int, optional

The number of times the Latin square should be repeated (default=1). Duplication occurs *after* constructing the square.

balanced : bool, optional

If True (the default), first-order order effects will be balanced Each condition will appear the same number of times immediately before and immediately after every other condition. Balanced latin squares can only be constructed with an even number of conditions.

uniform : bool, optional

If True (default is False), the Latin square will be randomly sampled from a uniform distribution of Latin squares of size NxN. Otherwise, the sampling will be biased. The construction of balanced, uniform Latin squares is not implemented.

Notes

The algorithm for computing unbalanced Latin squares is not very efficient. It is not recommended to construct unbalanced, uniform Latin squares of order above 5; for non-uniform, unbalanced Latin squares it is safe to go up to an order of 10. Higher than that, computation times increase rapidly.

The algorithm for computing balanced Latin squares is fast only because it is not robust; it is very biased and only samples from the same limited set of balanced Latin squares. However, this is usually not an issue. For more implementation details, see `latin_square` and `balanced_latin_square`.

`first_pass`(*conditions*)

Handle operations that should only be performed once, initializing the object before ordering conditions. For *`LatinSquare`*, the square is constructed.

Parameters **conditions** : sequence of dict

A list of conditions, where each condition is a dictionary mapping IV names to IV values.

Returns **iv_name** : str or tuple

The name of the IV to be created one level up, `'latin_square_row'`.

iv_values : tuple

Values of the IV to be created one level up, integers each corresponding to one row of the Latin square.

5.1.5 Indices and tables

- genindex
- modindex
- search

Python Module Index