Programming for Psychology in Python

Data Analysis and Visualisation

Introduction

Welcome to this set of lessons on using Python for data analysis and visualisation in the context of psychology. The aim of these lessons is to provide you with a set of skills that you can use to analyse data related to experiments and create publication-quality figures.

These lessons assume that you have completed the series on the Fundamentals.

What content will we be covering?

In this series of lessons, we will be looking at:

1. A new data type that is very important for data analysis (the *array*) and how we can save and load data.
2. How we can produce publication-quality figures in Python using the veusz package.
3. Using Python to produce descriptive statistics, do bootstrapping, and make boxplot and error bar figures.
4. Calculating and interpreting simple inferential statistics and producing scatter plots.
5. Using a computational simulation approach to perform power calculations, and make line and image figures.

How are these lessons organised?

Each lesson is focused around a key concept in the fundamentals of Python programming for data analysis and visualisation. The recommended approach is to follow along with the content while writing and executing the associated code.

You will also find that each lesson has an associated screencast. In each screencast, I narrate the process of coding and executing scripts related to the concept of the lesson. The purpose of these screencasts are to give a practical demonstration of the concepts of the lesson. They will typically cover much the same content as the written material. It is recommended that you both view the screencast and read through the written material.

You can also view the written material of all the lessons as a combined PDF.

The lessons also contain the occasional *tip*, which are small concepts that give additional suggestions on the relevant content (sometimes highlighting very important concepts). An instructive example is:

Tip: It is best to watch the screencasts in HD resolution—otherwise the text is not very legible. If you don't have a fast enough network connection, you can copy the video (mp4) files from the course directory and view those rather than streaming.

Finally, the lessons also include exercises for you to try.

Back to top

Programming for Psychology in Python

Data Analysis and Visualisation

Arrays

Objectives

- Be able to create arrays using numpy functions.
- Know how to access subsets of array elements using indexing.
- Understand how operations are applied to both the items in an array and over items in an array.
- Be able to save/load arrays to/from disk.

Screencast

In this lesson, we will be introducing a critical data type for data analysis—the array.

We will be using an external Python package called numpy for our array functionality. As you may recall, we need to use the `import` command to make such additional functionality available to our Python scripts. For numpy, we do something slightly different; because we will be using it so much, it is conventional to shorten `numpy` to `np` in our code. We modify our usual `import` statement to be:

```
import numpy as np
```

This code imports the numpy functionality and allows us to refer to it as `np`. Typically, this line will appear at the top of all the Python scripts we will use in these lessons.

Creating arrays

There are quite a few ways of producing arrays. First, we will consider the array analogue of the `range` function that we used previously to produce a list. The equivalent function in numpy is `arange`:

```
import numpy as np

r_list = range(10)
print type(r_list), r_list

r_array = np.arange(10)
print type(r_array), r_array
```

```
<type 'list'> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
<type 'numpy.ndarray'> [0 1 2 3 4 5 6 7 8 9]
```

As you can see, both methods have produced a collection of integers from 0 through 9. However, they are different data types—the `range` function produces a `list`, whereas the `arange` function produces an `array`. This distinction is important, because the different data types have different functionality associated with them.

One of the most useful things about arrays is that they can have multiple dimensions. A frequently encountered form of data is a table, with a number of rows and a number of columns. We can represent such a structure by creating a two-dimensional array. For example, the `ones` function creates an array of a particular size with all elements having the value of 1—we can use this to create a data structure with 5 rows and 3 columns.

```
import numpy as np

data = np.ones(shape=[5, 3])

print data
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Here, we've provided the `ones` function with a keyword argument called `shape`, which is a list that specifies the number of items along each dimension. As you can see, it creates a structure with 5 rows and 3 columns. You can see how this would be a useful data structure if you think that the rows might represent individual participants in an experiment and the columns might 3 conditions in a within-subject design.

Once created, we can access various useful properties of the array. For example, `.shape` returns a representation of the number of items along each dimension of the array, `.size` returns the total number of items in the array, and `.ndim` returns the number of dimensions in the array:

```
import numpy as np

data = np.ones(shape=[5, 3])

print data.shape
print data.size
print data.ndim
```

```
(5, 3)
15
2
```

Indexing arrays

We can access the items in an array using similar techniques to what we used with lists:

```
import numpy as np

r_array = np.arange(10)

print r_array[:5]

print r_array[::-1]
```

```
[0 1 2 3 4]
[9 8 7 6 5 4 3 2 1 0]
```

Indexing becomes more advanced when we have arrays with more than one dimension. Here, we will use another function to generate a multidimensional array—the numpy equivalent of `random.random` that we encountered earlier. We can access individual items in the array by separating the dimensions by a comma:

```
import numpy as np

data = np.random.random(size=[5, 3])

print data

# first row, first column
print data[0, 0]
# second row, first column
print data[1, 0]
# second row, last column
print data[1, -1]
```

```
[[ 0.77621576  0.01362286  0.21022832]
 [ 0.4305377   0.4401823   0.39921627]
 [ 0.14731176  0.39218311  0.69335116]
 [ 0.91274555  0.51841469  0.34363126]
 [ 0.27164045  0.43893888  0.49131077]]
0.776215764144
0.430537695746
0.399216272668
```

Importantly, we can also access all the items along a particular dimension:

```
import numpy as np

data = np.random.random(size=[5, 3])

print data

# all rows, first column
print data[:, 0]
# first row, all columns
print data[0, :]
```

```
[[ 0.88107544  0.69085362  0.62455685]
 [ 0.36892066  0.31793459  0.23899498]
 [ 0.62461964  0.76723846  0.33082585]
 [ 0.10430936  0.43618487  0.69452131]
 [ 0.39146278  0.04370793  0.1331806 ]]
[ 0.88107544  0.36892066  0.62461964  0.10430936  0.39146278]
[ 0.88107544  0.69085362  0.62455685]
```

We can also extract items using arrays of boolean values. For example, if we use a `>` operator on an array, it returns an array of booleans. If we then use this boolean array to index the data, it returns those items where the corresponding item in the boolean array is `True`:

```
import numpy as np

data = np.random.random(10)

print data

gt_point_five = data > 0.5

print gt_point_five

print data[gt_point_five]
```

```
[ 0.21893104  0.63712749  0.16704687  0.47526662  0.79358568  0.88655295
  0.09881262  0.40694586  0.58077379  0.62102438]
[False  True False False  True  True False False  True  True]
[ 0.63712749  0.79358568  0.88655295  0.58077379  0.62102438]
```

Operations on arrays

We can use the conventional maths operators with arrays where, unlike lists, they operate on each item in the array. For example:

```
import numpy as np

data = np.ones(4)

print data + 1
print data * 3
print data - 2
```

```
[ 2.  2.  2.  2.]
[ 3.  3.  3.  3.]
[-1. -1. -1. -1.]
```

We can also use operators that operate *over* items in an array. For example, we could add together all the items in an array:

```
import numpy as np

data = np.ones(4)

print data
print np.sum(data)
```

```
[ 1.  1.  1.  1.]
4.0
```

When applied to multidimensional arrays, such functions typically can be given an `axis` argument. This argument specifies the axis over which the operation is applied. For example, to sum over rows and columns:

```
import numpy as np

data = np.ones([4, 3])
data[1, :] = 2
data[2, :] = 3
data[3, :] = 4

print data

print "Rows:"
print np.sum(data, axis=0)

print "Columns:"
print np.sum(data, axis=1)
```

```
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 3.  3.  3.]
 [ 4.  4.  4.]]
Rows:
[ 10.  10.  10.]
Columns:
[  3.   6.   9.  12.]
```

Loading and saving arrays from/to disk

When using one or two dimensional arrays, a straightforward way to load and save data is in the form of a text file. This can be opened with any editor, and maximises the interoperability of the data with other programs. To do so, we use `np.savetxt` and `np.loadtxt`. For example, we can save some random data to a text file and the load it back in again to verify its contents have not changed:

```
import numpy as np

data = np.random.random([3, 2])

print data

np.savetxt("data.txt", data)

saved_data = np.loadtxt("data.txt")
```

```
print saved_data
```

```
[[ 0.28337331  0.87713617]
 [ 0.74152936  0.75838445]
 [ 0.8588712   0.65192374]]
[[ 0.28337331  0.87713617]
 [ 0.74152936  0.75838445]
 [ 0.8588712   0.65192374]]
```

Tip: One potentially important argument to `loadtxt` and `savetxt` is `delimiter`, which indicates what is used to separate the columns. If you've heard of a CSV file, that stands for "comma separated values". Another popular variant is TSV, "tab separated values". The default for `savetxt` and `loadtxt` is a space separating the columns, so you would need to set the `delimiter` argument if using a CSV or TSV file.

If the array has more than two dimensions, then saving as a plain text file often isn't practical. In such circumstances, you can use `np.save` and `np.load` with array files, which are typically given the extension `.npy`.

Exercises

1. Another function for creating an array is `np.array`. Use this function to create an array from a list, and from a list of lists.
2. Investigate the function `np.logical_and` and use it to select the items in an array generated by `np.random.random(10)` that are between `0.2` and `0.6`.
3. After having generated a three-dimensional array, how would you sum over the first and last dimension? Hint: two operations may be required.
4. It is good practice to include a 'header' in a text file that stores an array, which gives a readable description of the details of the array. How would you add such a header when using `savetxt`? Hint: investigate the arguments to the `savetxt` function.

Programming for Psychology in Python

Data Analysis and Visualisation

Creating figures

Objectives

- Be able to create the framework for a figure using Python and veusz.
- Create a histogram.
- Be able to change the formatting details of a figure.
- Understand the different figure file formats and how to save figures.

Screencast

In this lesson, we will be introducing the veusz package and using it to make publication-quality figures. We'll also start thinking about what makes a figure "publication-quality".

Setting up a figure

First, we need to understand how we go about constructing a figure using veusz.

To add the functionality of veusz to Python, we need to import it. For veusz, we actually need to import a subpackage called `embed`. We do this by:

```
import veusz.embed
```

Then, we need to create a veusz window that we can draw to. We do that using the `veusz.embed.Embedded` function, which takes an argument that sets the window title. This isn't important, but we'll set it to `veusz`. So that we can see what we have drawn, we then ask veusz to keep the window open until we close it.

```
import veusz.embed

embed = veusz.embed.Embedded("veusz")

# do all our drawing and formatting in here

embed.WaitForClose()
```

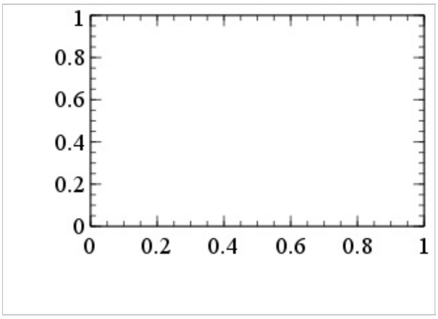
At this point, all we have is a blank window. We begin by adding a "page"—a canvas on which we can draw our figure. It is important to start thinking about what the figure is going to look like, because this will affect how large we make the page. Let's assume we're preparing a figure for publication in a journal. Typically, these can occupy either one or two columns of a printed page—corresponding to a figure width of either 8.4cm or 17.5cm. Here, we'll make a one-column figure that is slightly wider than high. We do this by setting the `.width.val` and `.height.val` properties of the page to their respective quantities, specified as a string and with the units.

```
import veusz.embed

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

embed.WaitForClose()
```

Now we have our page, we can add a graph to it. When doing this, we also use the argument `autoadd=False` to indicate that we will create the axes in the graph ourselves, which we then go ahead and do.

```
import veusz.embed

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.WaitForClose()
```

This creates the basic organisation of our figure:



Specifying the data to show in the figure

In this lesson, we're going to create a histogram. We have been using random numbers quite a bit so far, so let's look at the distribution of samples drawn from a Poisson distribution (the details of this distribution are not important—we use it here because it involves integers, which are easier to use in histograms). We can draw such samples using the `np.random.poisson` function. Let's draw 100 samples from the distribution. We can do this by the following code (for clarity, this code will be presented in isolation from the figure code. We will fold it back in later):

```
import numpy as np

n_samples = 100
```

```
samples = np.random.poisson(lam=4, size=n_samples)
```

Now, for our histogram we need the bin locations on the x axis and the bin counts on the y axis. But all we have at the moment is the array of samples. We can use the function `np.bincount` to count how many samples correspond to the range of integers present in the samples.

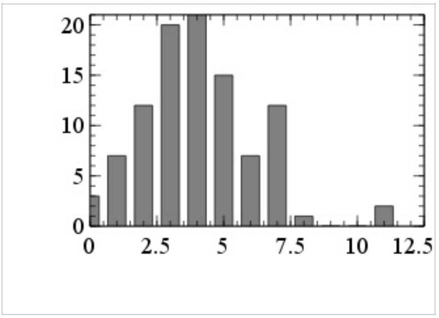
```
import numpy as np

n_samples = 100

samples = np.random.poisson(lam=4, size=n_samples)

bin_counts = np.bincount(samples)

bins = np.arange(np.max(samples) + 1)
```

Connecting the data to the figure

Now we have our data, we can go ahead and depict it in our figure. Accordingly, we will fold the code above back into our main figure code. Then we will create a bar graph, as is conventionally used to display this kind of count data.

We create a bar graph by first using the `Add` function of our `graph` variable, with the argument `"bar"` to indicate a bar graph. Then, we need to bind our two pieces of data (the bin locations and the bin counts) to the graph. First, we tell veusz about the data using the `SetData` method, which takes as arguments a name that we can refer to it by and the data array. Then, we set the `.posn.val` and `.lengths.val` properties of our `bar` variable to the names we gave to the bin location and bin count data, respectively. After this, we can see that we are getting somewhere!

```
import veusz.embed
import numpy as np

n_samples = 100

samples = np.random.poisson(lam=4, size=n_samples)

bin_counts = np.bincount(samples)

bins = np.arange(np.max(samples) + 1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

bar = graph.Add("bar")

embed.SetData("bins", bins)
embed.SetData("bin_counts", bin_counts)

bar.posn.val = "bins"
bar.lengths.val = "bin_counts"

embed.WaitForClose()
```

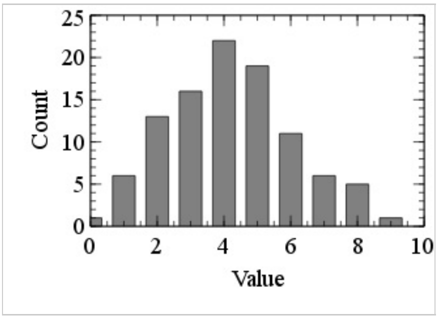


Improving the figure's appearance

Now let's work to add in the details on the figure and to improve its general appearance. Note that since we're drawing random samples, the distribution that is shown in subsequent figures will change slightly.

First, let's set the axis labels. We don't know what the samples represent in this example, so let's just give them generic names:

```
import veusz.embed
import numpy as np

n_samples = 100

samples = np.random.poisson(lam=4, size=n_samples)

bin_counts = np.bincount(samples)

bins = np.arange(np.max(samples) + 1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

bar = graph.Add("bar")

embed.SetData("bins", bins)
embed.SetData("bin_counts", bin_counts)

bar.posn.val = "bins"
bar.lengths.val = "bin_counts"

x_axis.label.val = "Value"
y_axis.label.val = "Count"

embed.WaitForClose()
```



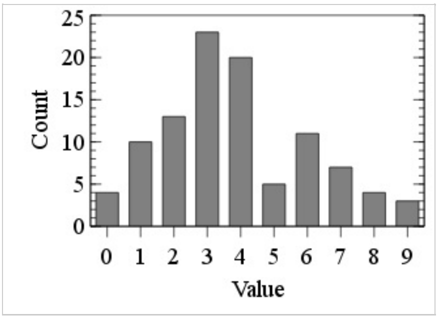
Now let's make some changes to the x axis. Looking at the above, there are a few changes that could be useful in improving the clarity of the figure:

- The minimum value on the axis could be reduced a bit to accommodate the bar at 0.
- The maximum value on the axis could be increased a bit to accommodate the bar at the largest bin.
- The minor ticks don't add much, so let's remove them.
- There is enough space to include all the major tick labels, so let's add them.
- There is not much to be gained by having the additional axis at the top of the figure, so let's remove it.
- The tick marks pointing upwards might interfere with our ability to see low values, so let's point them downwards instead.

We can implement such changes as follows:

```
import veusz.embed
import numpy as np

n_samples = 100

samples = np.random.poisson(lam=4, size=n_samples)
```

```
bin_counts = np.bincount(samples)

bins = np.arange(np.max(samples) + 1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

bar = graph.Add("bar")

embed.SetData("bins", bins)
embed.SetData("bin_counts", bin_counts)

bar.posn.val = "bins"
bar.lengths.val = "bin_counts"

x_axis.label.val = "Value"
y_axis.label.val = "Count"

# veusz can be picky about its numbers - here, we get Python to convert the
# result of the max function to a decimal (called a float)
x_axis.min.val = -0.5
x_axis.max.val = float(np.max(bins) + 0.5)
x_axis.MinorTicks.hide.val = True

# veusz requires the manual tick locations to be specified as a list of
# decimals, whereas we currently have them as a numpy array
x_axis.MajorTicks.manualTicks.val = bins.tolist()

x_axis.autoMirror.val = False
x_axis.outerticks.val = True

embed.WaitForClose()
```



Our figure is looking better, but there are a few new and potentially tricky aspects of the above code that we will need to review.

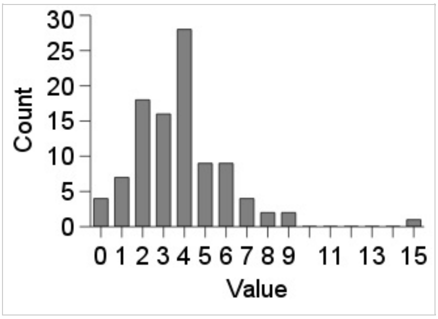
First, notice that we've used a new function, `float` when specifying the maximum of the x axis. This is because veusz can be rather picky about the type of data you give it. Because we were using numpy arrays, we have to explicitly convert the output into a Python decimal number. Such values are called "floats".

Second, we've used a new function attached to a numpy array, `tolist`. Unsurprisingly, this converts the array into a Python list data type, which is what veusz is expecting for this component of the graph.

Next, we will consider the y axis. Again, we will apply many of the same modifications as we did for the x axis. We will also remove the frame around the figure, and change the typeface of the text to a sans-serif variant (such as Arial).

```
import veusz.embed
import numpy as np

n_samples = 100

samples = np.random.poisson(lam=4, size=n_samples)

bin_counts = np.bincount(samples)
```

```
bins = np.arange(np.max(samples) + 1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

bar = graph.Add("bar")

embed.SetData("bins", bins)
embed.SetData("bin_counts", bin_counts)

bar.posn.val = "bins"
bar.lengths.val = "bin_counts"

x_axis.label.val = "Value"
y_axis.label.val = "Count"

# veusz can be picky about its numbers - here, we get Python to convert the
# result of the max function to a decimal (called a float)
x_axis.min.val = -0.5
x_axis.max.val = float(np.max(bins) + 0.5)
x_axis.MinorTicks.hide.val = True

# veusz requires the manual tick locations to be specified as a list of
# decimals, whereas we currently have them as a numpy array
x_axis.MajorTicks.manualTicks.val = bins.tolist()

x_axis.autoMirror.val = False
x_axis.outerticks.val = True

y_axis.MinorTicks.hide.val = True
y_axis.autoMirror.val = False
y_axis.outerticks.val = True

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:
    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

embed.WaitForClose()
```



Note the use of a `for` loop in the above code. Because we wanted to apply the same statements to both the x and y axes, we can create a list and loop over it rather than type the commands separately for each axis.

Saving the figure

Now we have a nice looking figure, the final step is to save it into a format that we can include in our document. The primary decision we need to make here is whether to save it in a vector or bitmap format. A vector-based format (such as PDF and SVG) stores the component as shapes, which can often allow for smaller file sizes and intact resolution at multiple scales (i.e. zoom friendly). The downsides are that not all software supports this format (Word, notably) and it can struggle with some figure aspects such as transparency. The alternative is a bitmap-based format (such as PNG and TIFF) that creates a pixel-by-pixel representation. When created for publication, these files need to be created with a high DPI (dots per inch) such as 600.

Let's save a few different formats. We can do this using `embed.Export` at the end of our code, as follows:

```
embed.Export("poisson.pdf", backcolor="white")

for dpi in [72, 600]:
    embed.Export("poisson_{dpi:n}.png".format(dpi=dpi), backcolor="white", dpi=dpi)
```

A new element in the above code is the use of string formatting via the `.format` function. This allows us to embed string representations of variables within other strings. By using the braces (`{` and `}`), we are saying that we are going to specify a variable called `dpi` that is a number (`n`). The `format` function then inserts it into the string, allowing us to embed the dpi value in the filename.

Exercises

1. What does the property `bar.barfill.val` affect?
2. Create a blank Microsoft Word document and insert the two PNG images that we created (N.B. we did not do this step in the screencast; see above for details). Can you see any differences between the two images? Then, open the PDF file in Adobe Acrobat. Try zooming in a long way and see what happens. Then try a similar zooming on the images in the Microsoft Word document.
3. Make a histogram for 1000 samples from a Gaussian distribution with a mean of 0 and a standard deviation of 1, using 15 bins in the range from -3 to +3 (Hint: investigate the `np.histogram` function). Then, apply your judgement to improve the figure's appearance. N.B. You might find this exercise to be particularly challenging.

Back to top

Programming for Psychology in Python

Data Analysis and Visualisation

Descriptive statistics—calculation and visualisation

Objectives

- Know how to create a figure with multiple panels.
- Be able to calculate basic descriptive statistics.
- Be able to use boxplots to visualise distributions.
- Use 'bootstrapping' to compute confidence intervals.
- Be able to create error plots.

Screencast

In this lesson, we will be investigating how we can use Python to calculate basic (but important) descriptive statistics—including bootstrapping approaches that leverage the power of programming. We will also be learning more about figure creation, including how to accommodate the common requirement for a figure to have multiple panels and about two new figure types (boxplots and error plots).

A warning that this is a long lesson and covers a fair bit of ground.

Generating some data

Typically, the process would begin with the loading of data produced from an experiment. Here, to simplify things, we will instead simulate the data that might have been produced by an experiment. In particular, we will simulate the outcome of a between-subjects experiment with 3 conditions/groups and 30 participants per group. In our data, the three conditions are represented by normal distributions with different means and standard deviations. To make the subsequent code examples more concise, we will save the generated data to a text file that we can then load as required.

```
import numpy as np

# how many (simulated) participants in each group
n_per_group = 30

# mean of each group's distribution
group_means = [1.0, 2.0, 3.0]
# set the standard deviation of each group to be half of its mean
group_stdevs = [0.5, 1.0, 1.5]

n_groups = len(group_means)

# create an empty array where rows are participants and columns are groups
data = np.empty([n_per_group, n_groups])
# fill it with a special data type, 'nan' (Not A Number)
data.fill(np.nan)

for i_group in range(n_groups):

    group_mean = group_means[i_group]
    group_stdev = group_stdevs[i_group]

    # generate the group data by sampling from a normal distribution with the
    # group's parameters
    group_data = np.random.normal(
        loc=group_mean,
        scale=group_stdev,
        size=n_per_group
    )

    data[:, i_group] = group_data

# check that we have generated all the data we expected to
assert np.sum(np.isnan(data)) == 0

# let's have a look at the first five 'participants'
print data[:5, :]

# and save
data_path = "prog_data_vis_descriptives_sim_data.tsv"

np.savetxt(
    data_path,
    data,
    delimiter="\t",
    header="Simulated data for the Descriptives lesson"
)
```

```
[[ 1.04821267  0.65600117  5.74658475]
 [ 0.43324506  0.99387692  0.88523461]
 [ 0.20825822  1.85549866  3.15891402]
 [ 2.09221461  2.25089713  5.32282943]
 [ 0.1451846   1.36716601  2.93712124]]
```

Most of the above code will be familiar, with the exception of a new statement—`assert`. The `assert` statement is quite simple: if the following boolean does not evaluate to True, then Python raises an error. Here, we are using it to verify that we have filled up our array like we expected to. The function `np.isnan` returns a boolean array with elements having the value of True where the corresponding element in the input array is `nan`, and False otherwise. Then, we use `np.sum` to count up all the `True` values. If we have filled up our data array correctly, then there won't be any `nan` elements so the sum will be zero and we can happily proceed with the rest of the code. But if there are `nan` values in there, we've made a mistake and the program will notify us. The `assert` statement is a very useful way of verifying that aspects of the code are doing what you think, and it is often good practice to include `assert` statements quite liberally.

Visualising the distributions

The first thing we will do is take a look at the distributions using the histogram technique we encountered in the previous lesson. However, we have three conditions here—we could think about including them all in the one histogram, but it might get a bit crowded. Instead, we will create a figure with multiple panels, each showing the histogram for one condition.

One of the primary purposes of generating such a figure is to be able to compare the distributions across the three conditions. Hence, a vertical stacking of panels would seem to be better than horizontal stacking because it would allow a direct comparison of density by looking down the page. So let's generate a one-column figure (8.4cm wide) with three vertical panels (16cm high).

First, we will setup up the framework for our figure like we have done previously:

```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

embed.WaitForClose()
```

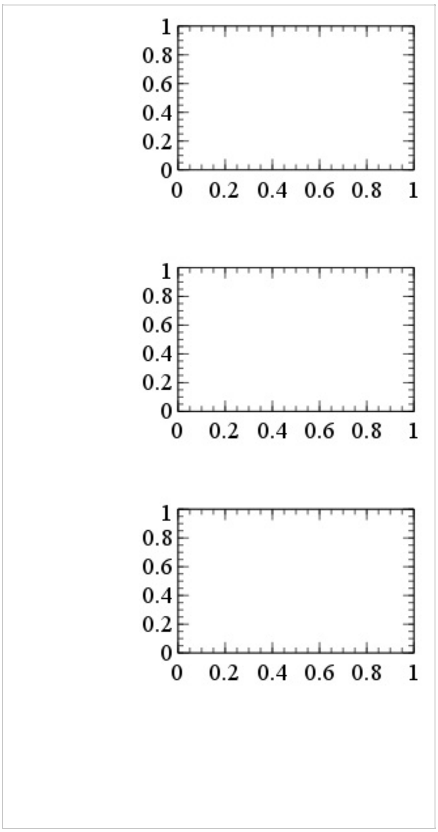
Previously, we added a `graph` element directly to the `page`. Here, we want to have multiple panels so we first add a `grid` element to the page. Such a grid has two particularly important properties, `rows` and `columns`. Here, we want just the one column and three rows (to have our three panels stacked vertically).

```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = int(n_cond)
grid.columns.val = 1

embed.WaitForClose()
```

Now, rather than adding the `graph` to the `page`, we add it to the `grid`. Consecutive `graph`s will then appear in different panels:

```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")
```

```
page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = int(n_cond)
grid.columns.val = 1

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

embed.WaitForClose()
```



Ok, now we are getting somewhere! We have the basic framework for our three panels, now we need to 'fill' each panel.

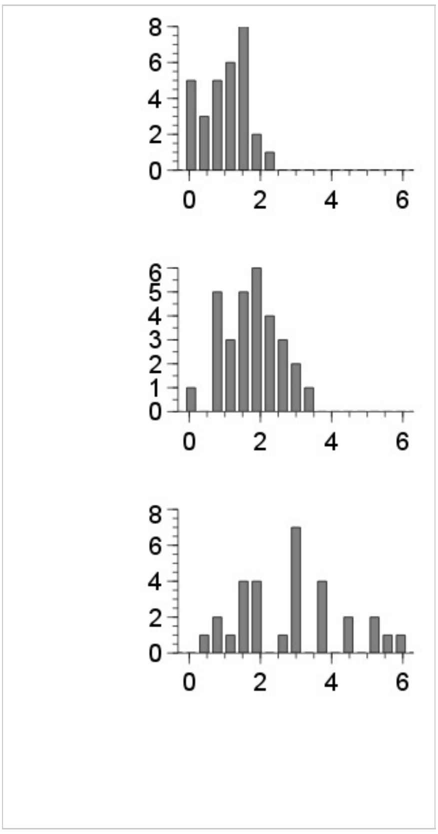
We are going to be making histograms, and the first important decision we need to make are the positions of the 'bins'. Because we want to compare across the three panels, it is desirable for them to all have the same bins. One plausible way of deciding the bins is to use the minimum and maximum values across all conditions as the bin extremes. We'll add a small amount of padding as well to give them a bit of room at the edges. Then, we will use the `np.linspace` to generate a series of values between the extremes. Then, because we have defined the bin edges but we want to plot the bin centres, we work out the bin size (as the difference between two adjacent edges) and add half that size to each edge. Finally, we make veusz aware of our bin centres, which will apply to each of the figure panels. Because the last bin edge doesn't contribute to the counts, we don't include the last bin.

```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")
```

```
page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = int(n_cond)
grid.columns.val = 1

min_val = np.min(data)
max_val = np.max(data)

bin_pad = 0.5

n_bins = 20

bin_edges = np.linspace(
    min_val - bin_pad,
    max_val + bin_pad,
    n_bins,
    endpoint=True
)

bin_delta = bin_edges[1] - bin_edges[0]

bin_centres = bin_edges + (bin_delta / 2.0)

embed.SetData("bin_centres", bin_centres[:-1])

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

embed.WaitForClose()
```

Now, we can calculate the histogram for each condition and use the same techniques that we used previously to plot the histograms as bar graphs. We will also set some properties of the plots to improve their appearance, like we have done previously.

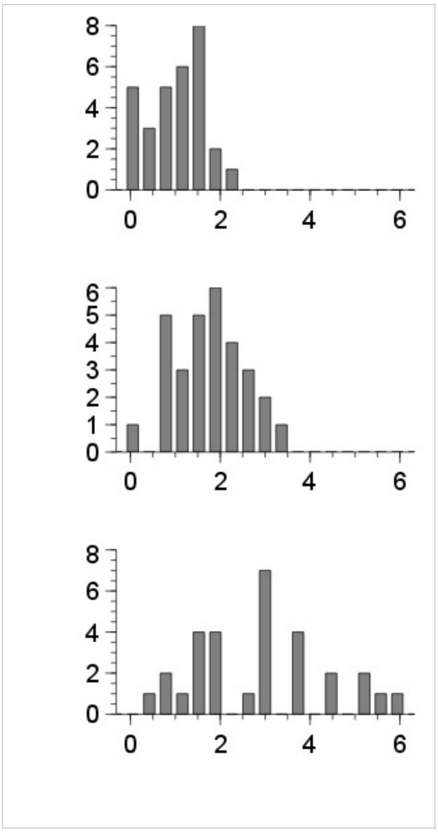
```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = int(n_cond)
grid.columns.val = 1

min_val = np.min(data)
max_val = np.max(data)

bin_pad = 0.5

n_bins = 20

bin_edges = np.linspace(
    min_val - bin_pad,
    max_val + bin_pad,
    n_bins,
    endpoint=True
)

bin_delta = bin_edges[1] - bin_edges[0]

bin_centres = bin_edges + (bin_delta / 2.0)

embed.SetData("bin_centres", bin_centres[:-1])

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

    hist_result = np.histogram(data[:, i_cond], bin_edges)

    bin_counts = hist_result[0]

    bin_counts_str = "bin_counts_{i:d}".format(i=i_cond)

    embed.SetData(bin_counts_str, bin_counts)

    bar = graph.Add("bar")

    bar.posn.val = "bin_centres"
    bar.lengths.val = bin_counts_str

    graph.Border.hide.val = True

    typeface = "Arial"

    for curr_axis in [x_axis, y_axis]:

        curr_axis.Label.font.val = typeface
        curr_axis.TickLabels.font.val = typeface

        curr_axis.autoMirror.val = False
        curr_axis.outerticks.val = True

embed.WaitForClose()
```



Something that is apparent in the above output is that the left and bottom margins are unnecessarily large. We can change the margins using the `leftMargin` and `bottomMargin` properties of the `grid`:

```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = int(n_cond)
grid.columns.val = 1

grid.leftMargin.val = "0.5cm"
grid.bottomMargin.val = "0.5cm"

min_val = np.min(data)
max_val = np.max(data)

bin_pad = 0.5

n_bins = 20

bin_edges = np.linspace(
    min_val - bin_pad,
    max_val + bin_pad,
    n_bins,
    endpoint=True
)

bin_delta = bin_edges[1] - bin_edges[0]

bin_centres = bin_edges + (bin_delta / 2.0)

embed.SetData("bin_centres", bin_centres[:-1])

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

    hist_result = np.histogram(data[:, i_cond], bin_edges)

    bin_counts = hist_result[0]

    bin_counts_str = "bin_counts_{i:d}".format(i=i_cond)

    embed.SetData(bin_counts_str, bin_counts)

    bar = graph.Add("bar")

    bar.posn.val = "bin_centres"
    bar.lengths.val = bin_counts_str

    graph.Border.hide.val = True

    typeface = "Arial"

    for curr_axis in [x_axis, y_axis]:

        curr_axis.Label.font.val = typeface
        curr_axis.TickLabels.font.val = typeface

        curr_axis.autoMirror.val = False
        curr_axis.outerticks.val = True
```

```
embed.WaitForClose()
```



There are a couple of things that need improving about the figure, but it is looking pretty good. We will leave these last couple of tweaks to the exercises.

Calculating descriptive statistics

We have visualised our data distributions, and now we can look at how we can calculate descriptive statistics that capture particular aspects of such distributions. Such descriptive statistics are straightforward in Python. The key aspect of our usage of the relevant functions is to specify the `axis=0` keyword; this tells Python to perform the relevant operation over the first axis, which is 'participants' in our array structure.

```python
import numpy as np

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# central tendency
print "Mean:", np.mean(data, axis=0)
print "Median:", np.median(data, axis=0)

# dispersion
print "Standard deviation:", np.std(data, axis=0, ddof=1)
print "Range:", np.ptp(data, axis=0)
```

```
Mean: [ 1.00667778  1.73985714  2.91635409]
Median: [ 1.05751834  1.74094639  2.94083891]
Standard deviation: [ 0.56199216  0.79633187  1.5153004 ]
Range: [ 2.03432348  3.24119132  5.47348541]
```

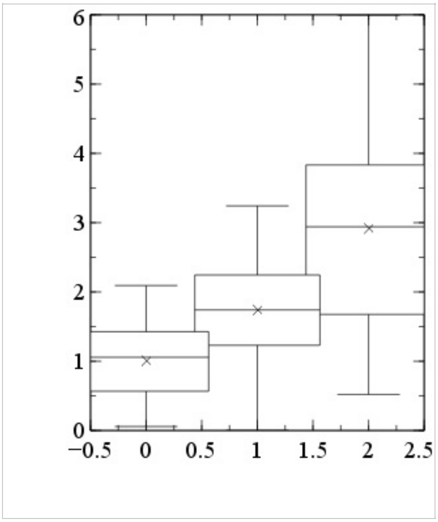
We generated our data from known distributions, and we can see the means and standard deviations follow our specifications pretty closely.

The names of the numpy functions are mostly intuitive with their functionality (with the exception of `ptp` for range; `ptp` is 'peak-to-peak'). One potentially mysterious argument is `ddof=1` for the standard deviation. If we look at the help for `np.std`, we can see that this corresponds to the value that N is reduced by in the denominator of the standard deviation calculation. With the default value of 0, this corresponds to the population standard deviation. Because we want the sample standard deviation, setting `ddof=1` corresponds to using `N - 1` in the denominator.

Visualising the distributions via boxplots

Boxplots are a very useful way of visualising distributions. We can create a boxplot in veusz by following the techniques we have used previously to create a graph, and then adding a `boxplot` element to the graph. We set the `values` property of the boxplot for each condition to the data array for that condition, and veusz calculates the relevant descriptive statistics and creates the boxplot. We also use the `posn` property to position each boxplot along the x axis corresponding to that condition's index.

```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "10cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

for i_cond in range(n_cond):

    box_str = "box_data_{i:d}".format(i=i_cond)

    embed.SetData(box_str, data[:, i_cond])

    boxplot = graph.Add("boxplot")

    boxplot.values.val = box_str
    boxplot.posn.val = i_cond

embed.WaitForClose()

embed.Export("d4a.png", backcolor="white")
```

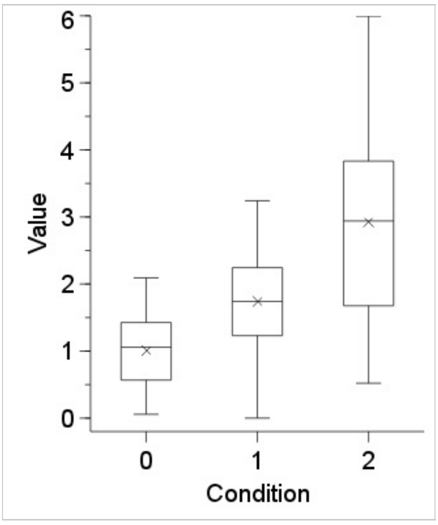


In each boxplot, the horizontal line is the median, the cross is the mean, the top and bottom of the 'boxes' are the 75th and 25th percentile, and the 'whiskers' extend 1.5 times the inter-quartile range beyond the 25th and 75th percentiles. Any values that lie outside the whiskers are marked individually as circles.

The current plot looks a bit ugly, though—let's spruce it up a bit. We will first use techniques that are mostly familiar by now.

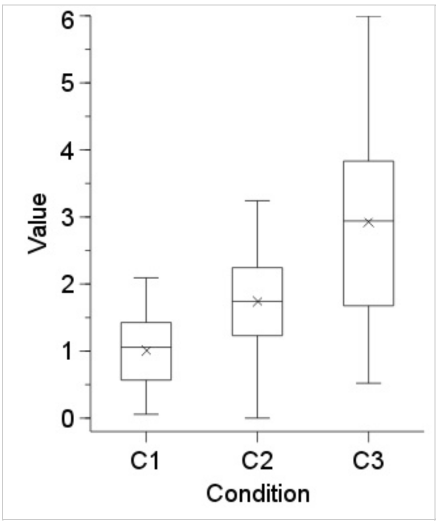
```
import numpy as np
```

```python
import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "10cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

for i_cond in range(n_cond):

    box_str = "box_data_{i:d}".format(i=i_cond)

    embed.SetData(box_str, data[:, i_cond])

    boxplot = graph.Add("boxplot")

    boxplot.values.val = box_str
    boxplot.posn.val = i_cond

    # reduce the 'width' of each box
    boxplot.fillfraction.val = 0.3

x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"

y_axis.min.val = float(np.min(data) - 0.2)
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



It is looking a lot better now, but one anomaly is that the x axis specifies numerical position values whereas they are really categorical condition labels. We can use strings instead of numbers in veusz by associating each boxplot with a `label` and telling the x axis to use such labels on the 'ticks':

```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "10cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

cond_labels = ["C1", "C2", "C3"]

for i_cond in range(n_cond):

    box_str = "box_data_{i:d}".format(i=i_cond)

    embed.SetData(box_str, data[:, i_cond])

    boxplot = graph.Add("boxplot")

    boxplot.values.val = box_str
    boxplot.posn.val = i_cond
    boxplot.labels.val = cond_labels[i_cond]

    # reduce the 'width' of each box
    boxplot.fillfraction.val = 0.3

x_axis.mode.val = "labels"
x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"

y_axis.min.val = float(np.min(data) - 0.2)
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



Computing confidence intervals

We are often interested in the means of our distributions. The typical way to calculate 95% confidence intervals (CIs) is via the standard error of the mean:

```
import numpy as np

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

# calculate standard error of each condition as the sample standard deviation
# divided by sqrt n
sem = np.std(data, axis=0, ddof=1) / np.sqrt(n_per_group)

print "SEMs: ", sem

for i_cond in xrange(n_cond):

    cond_mean = np.mean(data[:, i_cond])
    cond_sem = sem[i_cond]

    lower_ci = cond_mean - 1.96 * cond_sem
    upper_ci = cond_mean + 1.96 * cond_sem

    print "{i:d}: M = {m:.3f}, 95% CI [{lci:.3f}, {uci:.3f}]".format(
        i=i_cond + 1,
        m=cond_mean,
        lci=lower_ci,
        uci=upper_ci
    )
```

```
SEMs:  [ 0.10260526  0.14538964  0.27665474]
1: M = 1.007, 95% CI [0.806, 1.208]
2: M = 1.740, 95% CI [1.455, 2.025]
3: M = 2.916, 95% CI [2.374, 3.459]
```

Here, we will use an alternative non-parametric method called *bootstrapping* to calculate the CIs. This is a flexible technique that can be usefully applied to many data analysis situations. It is particularly interesting for our purposes because it requires a substantial amount of computation—yet is straightforward to implement once you know the fundamentals of programming.

The key to bootstrapping is to perform resampling with replacement. For example, generating the 95% CI for the mean of a given condition would involve randomly sample a group of participants with replacement—that is, a given participant could appear in the sampled data multiple times, or not at all. Let's first take a look at how we can do this:

```
import numpy as np

# generate subject indices from 0 to 9
```

```
subjects = range(10)

rand_subj_sample = np.random.choice(
    a=subjects,
    replace=True,
    size=10
)

print rand_subj_sample
```

```
[1 7 0 0 9 7 8 8 3 3]
```

As you can see in the above, we are randomly choosing with replacement a subset of the available participants. Once we have them, we can then pull out their data and compute a mean. Let's see it in action in the context of the first condition of the data we have been looking at:

```
import numpy as np

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

subjects = range(n_per_group)

rand_subj_sample = np.random.choice(
    a=subjects,
    replace=True,
    size=n_per_group
)

boot_data = np.mean(data[rand_subj_sample, 0])

print boot_data
```

```
0.96886989528
```

Bootstrapping involves performing this procedure many, many times. We will use 10,000 iterations here to establish our bootstrapped distribution. When we have calculated such a large number of such means, we can rank the values and look at the 2.5% and 97.5% values as our 95% confidence intervals. The function `scipy.stats.scoreatpercentile` is very handy for such a calculation:

```
import numpy as np

import scipy.stats

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

subjects = range(n_per_group)

n_boot = 10000

boot_data = np.empty(n_boot)
boot_data.fill(np.nan)

for i_boot in range(n_boot):

    rand_subj_sample = np.random.choice(
        a=subjects,
        replace=True,
        size=n_per_group
    )

    boot_data[i_boot] = np.mean(data[rand_subj_sample, 0])
```

```
cond_mean = np.mean(boot_data)
lower_ci = scipy.stats.scoreatpercentile(boot_data, 2.5)
upper_ci = scipy.stats.scoreatpercentile(boot_data, 97.5)

print "{i:d}: M = {m:.3f}, 95% CI [{lci:.3f}, {uci:.3f}]".format(
    i=1,
    m=cond_mean,
    lci=lower_ci,
    uci=upper_ci
)
```

```
1: M = 1.006, 95% CI [0.809, 1.199]
```

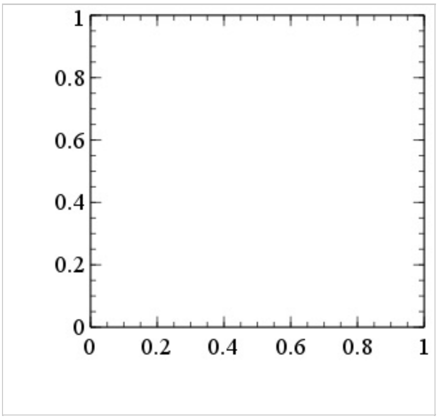
As you can see, it produces similar results to our previous method using the parametric approach.

Visualising condition means via error plots

Often a key figure when reporting research is a comparison of means across conditions, almost always with an indication of the uncertainty ('error') associated with each mean estimate. Here, we will plot the means and bootstrapped 95% CIs for each of our conditions. We will depict each mean as a point, with bars indicating the 95% CIs.

We begin by setting up our figure framework, as usual:

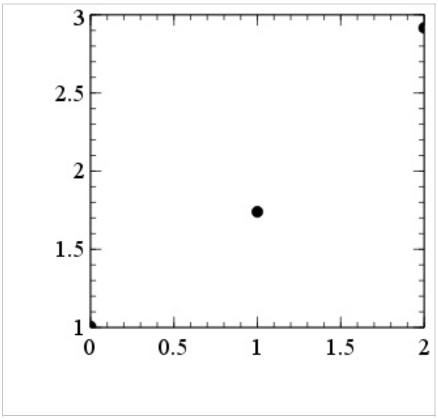
```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.WaitForClose()
```



Now, we are going to use a new graph type, `xy`, to plot our means:

```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
```

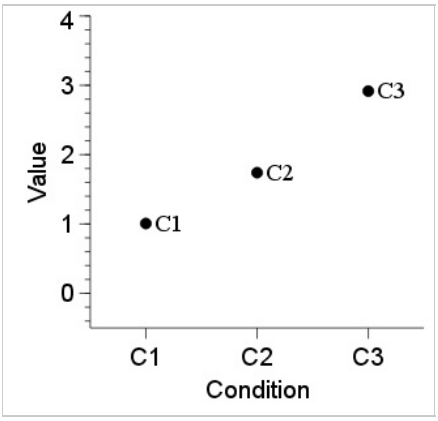
```
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

for i_cond in range(n_cond):

    cond_mean = np.mean(data[:, i_cond])

    data_str = "cond_{i:d}".format(i=i_cond)

    # cond_mean is a single value whereas SetData is expecting a list or array,
    # so pass a one-item list
    embed.SetData(data_str, [cond_mean])

    xy = graph.Add("xy")

    xy.xData.val = i_cond
    xy.yData.val = data_str

embed.WaitForClose()
```



Let's clean up the figure now, before we add in the CIs:

```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

cond_labels = ["C1", "C2", "C3"]
```

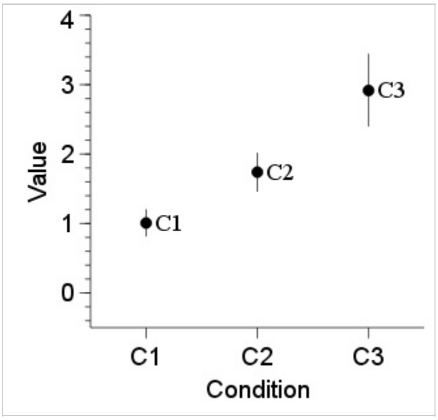
```
for i_cond in range(n_cond):

    cond_mean = np.mean(data[:, i_cond])

    data_str = "cond_{i:d}".format(i=i_cond)

    # cond_mean is a single value whereas SetData is expecting a list or array,
    # so pass a one-item list
    embed.SetData(data_str, [cond_mean])

    xy = graph.Add("xy")

    xy.xData.val = i_cond
    xy.yData.val = data_str
    xy.labels.val = cond_labels[i_cond]

x_axis.mode.val = "labels"
x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"
x_axis.min.val = -0.5
x_axis.max.val = 2.5

y_axis.min.val = -0.5
y_axis.max.val = 4.0
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



OK, now we can compute our 95% CIs and add them in to the figure. We do this by using an extra set of arguments to the `SetData` function: `poserr` (positive error) and `negerr` (negative error). These determine the regions above and below the data point that correspond to the 'error'.

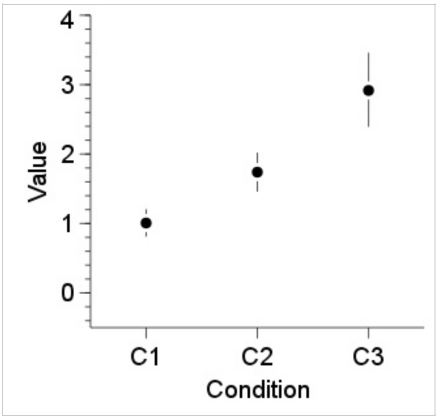
```
import numpy as np
import scipy.stats

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]
subjects = range(n_per_group)
```

```
embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

n_boot = 10000

cond_labels = ["C1", "C2", "C3"]

for i_cond in range(n_cond):

    cond_mean = np.mean(data[:, i_cond])

    # do bootstrapping
    boot_data = np.empty(n_boot)
    boot_data.fill(np.nan)

    for i_boot in range(n_boot):

        rand_subj_sample = np.random.choice(
            a=subjects,
            replace=True,
            size=n_per_group
        )

        boot_data[i_boot] = np.mean(data[rand_subj_sample, i_cond])

    lower_ci = scipy.stats.scoreatpercentile(boot_data, 2.5)
    upper_ci = scipy.stats.scoreatpercentile(boot_data, 97.5)

    pos_err = upper_ci - cond_mean
    neg_err = lower_ci - cond_mean

    data_str = "cond_{i:d}".format(i=i_cond)

    # cond_mean is a single value whereas SetData is expecting a list or array,
    # so pass a one-item list
    embed.SetData(
        data_str,
        [cond_mean],
        poserr=[pos_err],
        negerr=[neg_err]
    )

    xy = graph.Add("xy")

    xy.xData.val = i_cond
    xy.yData.val = data_str
    xy.labels.val = cond_labels[i_cond]

x_axis.mode.val = "labels"
x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"
x_axis.min.val = -0.5
x_axis.max.val = 2.5

y_axis.min.val = -0.5
y_axis.max.val = 4.0
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



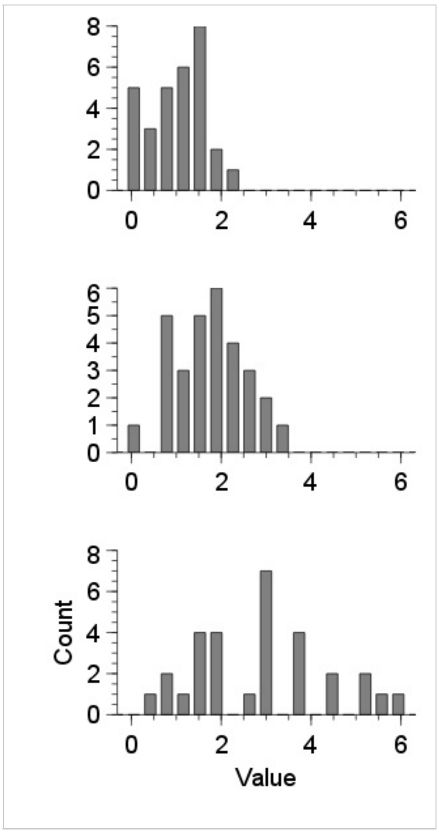
The figure is looking pretty good—let's make a couple of final tweaks to finish it off. First, the labels attached to each point can be useful sometimes but are probably unnecessary here. Second, let's add a white ring around each data point—mostly because it looks nice, but also to add a bit of separation between the point and its CI. Finally, let's make the points a bit bigger.

```
import numpy as np
import scipy.stats

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]
subjects = range(n_per_group)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

n_boot = 10000

cond_labels = ["C1", "C2", "C3"]

for i_cond in range(n_cond):

    cond_mean = np.mean(data[:, i_cond])

    # do bootstrapping
    boot_data = np.empty(n_boot)
    boot_data.fill(np.nan)

    for i_boot in range(n_boot):

        rand_subj_sample = np.random.choice(
            a=subjects,
            replace=True,
            size=n_per_group
        )

        boot_data[i_boot] = np.mean(data[rand_subj_sample, i_cond])

    lower_ci = scipy.stats.scoreatpercentile(boot_data, 2.5)
    upper_ci = scipy.stats.scoreatpercentile(boot_data, 97.5)

    pos_err = upper_ci - cond_mean
    neg_err = lower_ci - cond_mean

    data_str = "cond_{i:d}".format(i=i_cond)

    # cond_mean is a single value whereas SetData is expecting a list or array,
    # so pass a one-item list
    embed.SetData(
        data_str,
        [cond_mean],
        poserr=[pos_err],
        negerr=[neg_err]
    )

    xy = graph.Add("xy")

    xy.xData.val = i_cond
    xy.yData.val = data_str
    xy.labels.val = cond_labels[i_cond]

    xy.Label.hide.val = True
    xy.MarkerLine.color.val = "white"
    xy.MarkerLine.width.val = "2pt"

    xy.markerSize.val = "4pt"

x_axis.mode.val = "labels"
x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"
x_axis.min.val = -0.5
x_axis.max.val = 2.5

y_axis.min.val = -0.5
y_axis.max.val = 4.0
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```

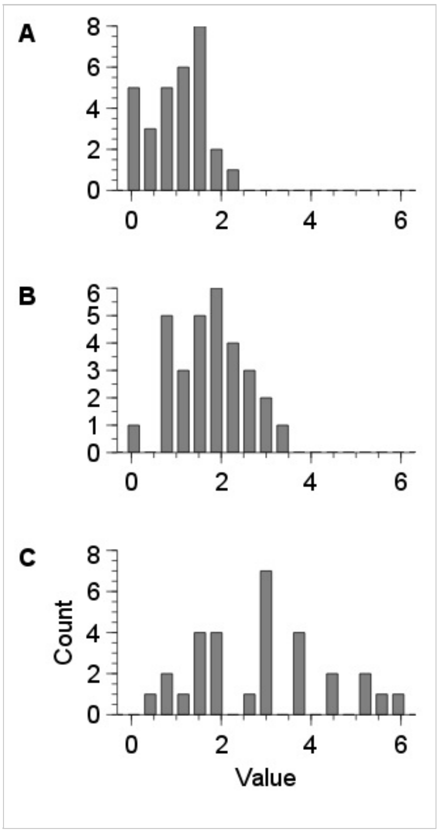


Exercises

1. It is sometimes reasonable for figures with multiple panels to only have the axis labels on one of the panels. Change the multi-panel histogram example to include axis labels ("Value" for the x axis, "Count" for the y axis), but only for the bottom panel.

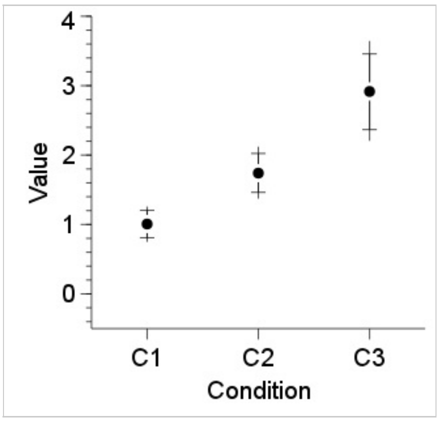


2. Figures with multiple panels often have a label attached to each panel ("A", "B", etc.). Attach a new veusz figure component, `label`, to each panel in the multi-panel histogram example. The panels should be named "A", "B", and "C" (investigate `label.val`) and positioned outside the horizontal axis in the top-left corner (investigate `xPos.val` and `yPos.val`; this might take some trial and error to get the positioning correct). Also, set the panel label font to be the same as the other text and make it bold (`Text.bold.val`).



3. We mentioned the descriptive statistics that veusz calculates for you when building a boxplot. Use Python functions to calculate such values (mean, median, 25th and 75th percentile, inter-quartile range, 1.5x the inter-quartile range, and any outliers) for the first condition. You might find the functions `np.mean`, `np.median`, and `scipy.stats.scoreatpercentile` to be useful.
4. Some suggest that it can be useful to indicate multiple CIs in error plots. Adapt the error plot to include both 95%

and 99% CIs for each data point. To differentiate them, set the 95% CIs to have 'capped' ends by setting `errorStyle.val` to be `"barends"` (the default is `bar`). Warning: this might be a challenging exercise.



Back to top

Damien J. Mannion, Ph.D. — School of Psychology — UNSW Australia

Programming for Psychology in Python

Data Analysis and Visualisation

Inferential statistics and scatter plots

Objectives

- Be able to calculate t-tests in Python.
- Use a simulation approach to understand false positives and multiple comparisons issues.
- Know how to create a figure to visualise relationships via scatter plots.
- Be able to evaluate correlations in Python.

Screencast

In this lesson, we will be investigating how we can use Python to calculate basic inferential statistics. We will also use programming to develop an increased intuition about some of the issues involved in statistics (particularly multiple comparisons considerations) and understand how to visualise relationships using scatter plots.

Comparing two independent means—t-tests

The first inferential statistic we will investigate is the independent-samples t-test, which is useful for comparing the means of two independent groups. As before, we will simulate the data that we might get from a between-subjects experiment with two conditions and 30 participants per condition. In our simulation, the effect size of the condition manipulation is 'large' (Cohen's d = 0.8).

```
import numpy as np

n_per_group = 30

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

data = np.empty([n_per_group, n_groups])
data.fill(np.nan)

for i_group in range(n_groups):

    data[:, i_group] = np.random.normal(
        loc=group_means[i_group],
        scale=group_sigmas[i_group],
        size=n_per_group
    )

assert np.sum(np.isnan(data)) == 0
```

Now that we have the data in an array form, computing the t-test is straightforward. We can use the function in `scipy.stats` called `scipy.stats.ttest_ind` (note that there is also `scipy.stats.ttest_rel` and `scipy.stats.ttest_1samp`). This function takes two arrays, `a` and `b`, and returns the associated t and p values from an independent-samples t-test.

```
import numpy as np

import scipy.stats

n_per_group = 30

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

data = np.empty([n_per_group, n_groups])
data.fill(np.nan)

for i_group in range(n_groups):

    data[:, i_group] = np.random.normal(
        loc=group_means[i_group],
        scale=group_sigmas[i_group],
        size=n_per_group
    )

assert np.sum(np.isnan(data)) == 0

result = scipy.stats.ttest_ind(a=data[:, 0], b=data[:, 1])

print "t: ", result[0]
print "p: ", result[1]
```

```
t:  -5.17157592947
p:  3.00846032319e-06
```

Demonstrating the problem with multiple comparisons

Now, we will use programming to hopefully develop more of an intuition about the operation of inferential statistics and why we need to think about the consequences of performing multiple comparisons. To simplify matters, we will switch to a one-sample design where we are looking to see if the mean is significantly different from zero. In contrast to our previous example, here the mean of the population that we sample from will actually be zero. We will perform a large number of such 'experiments' and see how often we reject this null hypothesis. Remember: because we are controlling the simulation, we know that there is no true difference between the population mean and 0.

```
import numpy as np

import scipy.stats
```

```python
n = 30

pop_mean = 0.0
pop_sigma = 1.0

# number of simulations we will run
n_sims = 10000

# store the p value associated with each simulation
p_values = np.empty(n_sims)
p_values.fill(np.nan)

for i_sim in range(n_sims):

    # perform our 'experiment'
    data = np.random.normal(
        loc=pop_mean,
        scale=pop_sigma,
        size=n
    )

    # perform the t-test
    result = scipy.stats.ttest_1samp(a=data, popmean=0.0)

    # save the p value
    p_values[i_sim] = result[1]

assert np.sum(np.isnan(p_values)) == 0
```

Ok, so now we have generated 10,000 p values from a situation in which we know that the population mean was not actually different from 0. We are interested in how many of those 10,000 'experiments' would yield a p value less than 0.05 and would lead to a false rejection of the null hypothesis. What would you predict? Let's have a look:

```python
import numpy as np

import scipy.stats

n = 30

pop_mean = 0.0
pop_sigma = 1.0

# number of simulations we will run
n_sims = 10000

# store the p value associated with each simulation
p_values = np.empty(n_sims)
p_values.fill(np.nan)

for i_sim in range(n_sims):

    # perform our 'experiment'
    data = np.random.normal(
        loc=pop_mean,
        scale=pop_sigma,
        size=n
    )

    # perform the t-test
    result = scipy.stats.ttest_1samp(a=data, popmean=0.0)

    # save the p value
    p_values[i_sim] = result[1]

assert np.sum(np.isnan(p_values)) == 0

# number of null hypotheses rejected
n_null_rej = np.sum(p_values < 0.05)

# proportion of null hypotheses rejected
prop_null_rej = n_null_rej / float(n_sims)

print prop_null_rej
```

```
0.0506
```

About 0.05—the t-test is doing its job and keeping the false positive rate at the p value that we specified.

Now, what about if we had two conditions instead of one, and in both cases we are interested in whether their means are significantly different from zero.

```
import numpy as np

import scipy.stats

n = 30

pop_mean = 0.0
pop_sigma = 1.0

# number of conditions
n_conds = 2

# number of simulations we will run
n_sims = 10000

# store the p value associated with each simulation
p_values = np.empty([n_sims, n_conds])
p_values.fill(np.nan)

for i_sim in range(n_sims):

    # perform our 'experiment'
    data = np.random.normal(
        loc=pop_mean,
        scale=pop_sigma,
        size=[n, n_conds]
    )

    for i_cond in range(n_conds):

        # perform the t-test
        result = scipy.stats.ttest_1samp(
            a=data[:, i_cond],
            popmean=0.0
        )

        # save the p value
        p_values[i_sim, i_cond] = result[1]

assert np.sum(np.isnan(p_values)) == 0

# determine whether, on each simulation, either of the two tests were deemed to
# be significantly different from 0
either_rej = np.any(p_values < 0.05, axis=1)

n_null_rej = np.sum(either_rej)

# proportion of null hypotheses rejected
prop_null_rej = n_null_rej / float(n_sims)

print prop_null_rej
```

```
0.0943
```

In the above code, we have used the `np.any` function to test whether either of the columns on a given row (`axis=1`) is `True`. We can see that the probability of falsely rejecting the null hypothesis on either test in a given 'experiment' has roughly doubled.

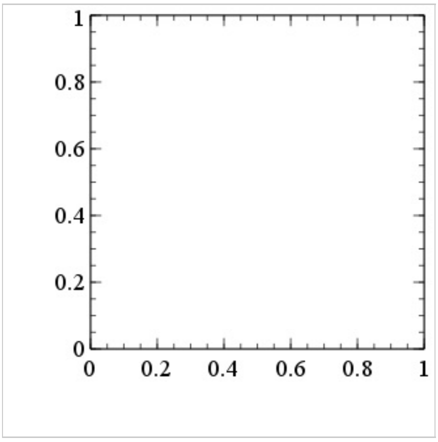
Relating two variables—correlations and their visualisation

Now we will move on to a situation in which you want to examine the relationship between two continuous variables via correlation analyses. Once again, we will start by generating some data that one might get from a real psychology experiment. Here, we have 30 participants from which we obtain two measurements each—in our experiment, the two measurements will have a 'large' negative correlation (r = -0.5).

```
import numpy as np

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)
```

Before we get started on how we can calculate correlation coefficients and test whether they are significantly different from zero, let's first look at how we can visualise such relationships. We will use a scatter plot to relate the data from the first condition (x axis) with the data from the second condition (y axis). First, we will set up the figure framework as we have done previously:

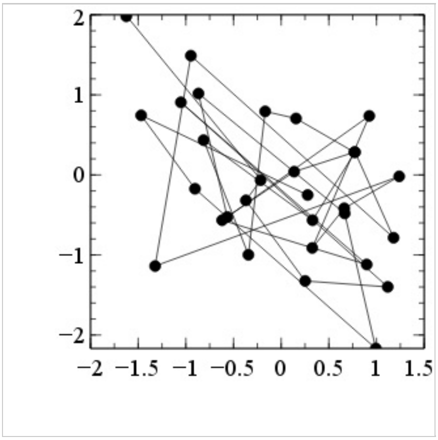
```
import numpy as np

import veusz.embed

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.WaitForClose()
```



Creating a scatter plot is actually a straightforward extension of a figure that we made in the last lesson, using the `xy` figure type. All we need to do is tell veusz about the data from the two conditions, create an `xy` figure element, and bind the data from the two conditions to the x and y axes:

```
import numpy as np

import veusz.embed

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
```

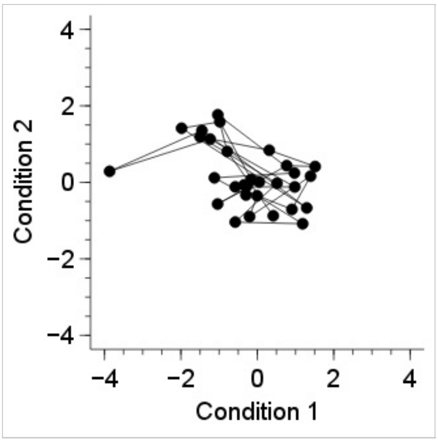
```
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

embed.WaitForClose()
```



Ok, we have made a bit of progress—but before it can start to look like a good scatter plot, let's first apply the formatting changes that we typically do. We also set a graph property that we haven't come across before, `aspect`, to 1.0—this makes the graph equally wide and tall, which is useful for scatter plots. We also set the axes to be symmetric about zero and the same for x and y.

```
import numpy as np

import veusz.embed

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)
```

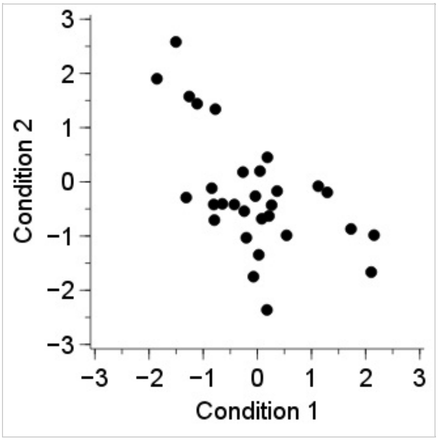
```
x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

embed.WaitForClose()
```



One problematic aspect of the current figure is that successive participants have their data points joined by a line. The ordering of participants is not relevant to the current analysis, so let's turn them off:

```
import numpy as np

import veusz.embed

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")
```

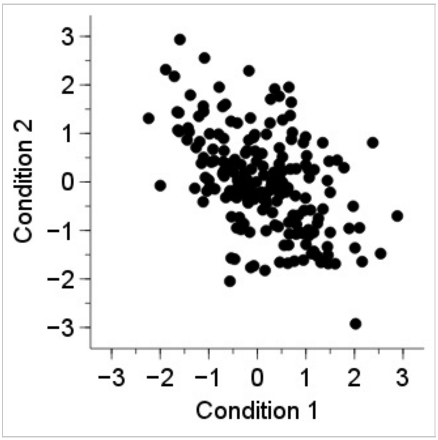
```
page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

embed.WaitForClose()
```



Now the figure is looking pretty good, but a final potential issue is that it is hard to see whether some of the dots overlap with one another. Let's increase the number of participants to make the problem more apparent:

```
import numpy as np

import veusz.embed

n = 200

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]
```

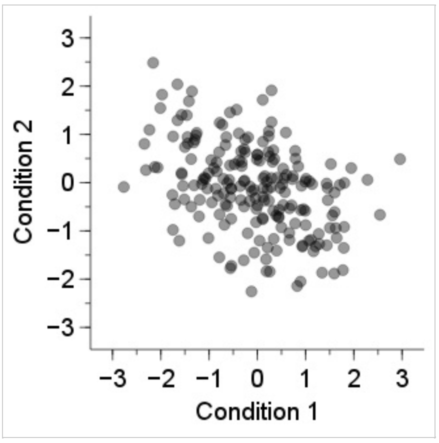
```
data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

embed.WaitForClose()
```



One solution that can work well is to make each of the dots a bit transparent—that way, it is more apparent when dots overlay one another. A downside of this approach is that figures with transparency can cause problems when exporting to PDF, so a bitmap-based approach is required.

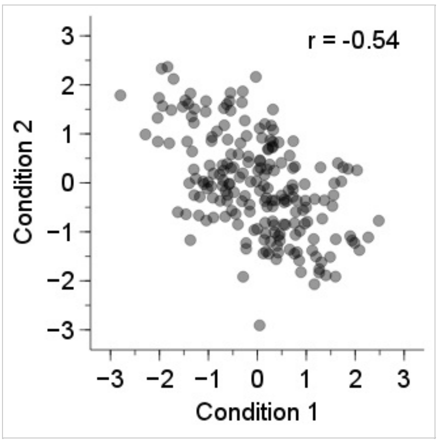
```
import numpy as np

import veusz.embed
```

```
n = 200

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True

xy.MarkerFill.transparency.val = 60
xy.MarkerLine.transparency.val = 60

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

embed.WaitForClose()
```



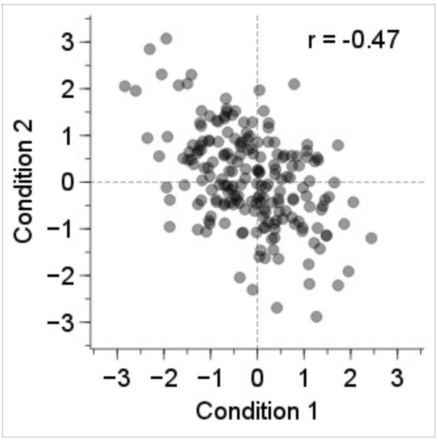
Now we have a good visualisation, let's go ahead and calculate the correlation coefficient (Pearson's correlation). To do so, we can use `scipy.stats.pearsonr`. This function takes two arguments, `x` and `y`, and returns a two-item list: the Pearson's correlation (r) and a p value. Let's then note the obtained r value directly on the figure.

```
import numpy as np

import scipy.stats

import veusz.embed

n = 200

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True

xy.MarkerFill.transparency.val = 60
xy.MarkerLine.transparency.val = 60

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:
    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

result = scipy.stats.pearsonr(x=data[:, 0], y=data[:, 1])
r = result[0]

r_fig = graph.Add("label")

r_fig.label.val = "r = {r:.2f}".format(r=r)
r_fig.xPos.val = 0.65
r_fig.yPos.val = 0.9
r_fig.Text.font.val = typeface

embed.WaitForClose()
```

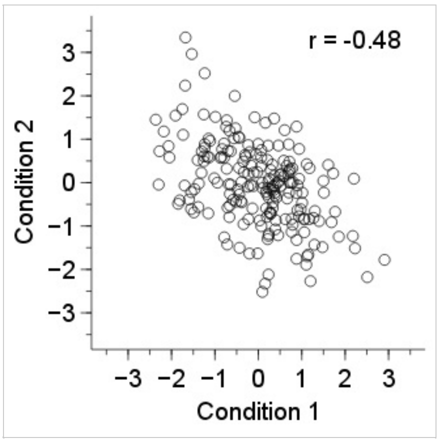


Exercises

1. How does the false positive rate change with increasing numbers of tests (hint: try changing `n_cond` in an example above)?
2. The Bonferroni procedure is a way of correcting for multiple comparisons by dividing the critical p value by the number of tests. Implement Bonferroni correction and check the effect on the false positive rate.
3. It can be useful to mark particular lines on the graph, such as the lines where x = 0 for all y and y = 0 for all x. Add such lines to the correlation figure, such as shown below (hint: `PlotLine.style.val = "dashed"` and `PlotLine.color.val = "grey"` might be useful).



4. We discussed using transparency to help make overlapping datapoints more visible. An alternative method would be to make the centre of each circle completely invisible, so that just the line around the edges is visible. Try this approach (hint: `xy.MarkerFill.hide.val`)—which do you prefer?



5. Use a bootstrapping approach to compute confidence intervals on a simulated -0.5 correlation with 30 participants. Compare the bootstrapped 95% confidence intervals with their parametric versions, calculated as `np.tanh(np.arctanh(r) +- 1.96 x 0.19245)`.

Programming for Psychology in Python

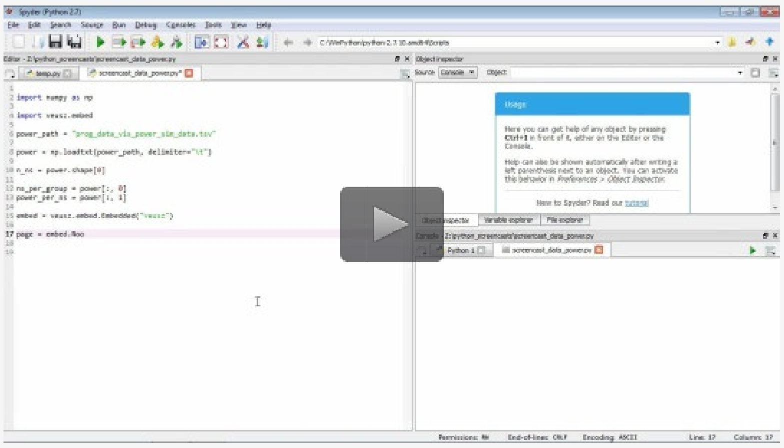
Data Analysis and Visualisation

Experimental design—power analysis and its visualisation

Objectives

- Be able to perform power calculations using computational simulation approaches.
- Know how to create and use line and image plots.

Screencast



```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

n_nc = power.shape[0]

ns_per_group = power[:, 0]
power_per_ns = power[:, 1]

embed = veusz.embed.Embedded("veusz")

page = embed.Roo
```

Power relates to the ability to detect the presence of a true effect and is an important component of experimental design. In this lesson, we will consider a general-purpose simulation approach to estimating the power of an experimental design. We will also investigate how we can visualise such data using line plots and two-dimensional image plots.

Calculating power given effect size and sample size

We will begin by considering a scenario in which we have an effect size and sample size in mind and we would like to know the associated power. For our example experiment, we will use a between-subjects design with two factors, 30 participants per group, and we will assume a 'large' effect size (Cohen's d = 0.8). You might recognise this design, as it was the same that we used in the previous lesson to demonstrate the operation of the independent-samples t-test. Here, we will determine the power of this test.

The key to determining power using a simulation approach is to again leverage the computational capacity that we have once we are able to write our own programs. We will perform a large number of simulated experiments, each time calculating our test statistic (independent samples t-test, in this case) and accumulating the number of times we reject the null hypothesis. Then, the power is simply the proportion of times that we are able to reject the null hypothesis (remembering that we control the population means and we *know* that there is a true difference).

```python
import numpy as np

import scipy.stats

n_per_group = 30

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

# number of simulations
n_sims = 10000

# store the p value for each simulation
sim_p = np.empty(n_sims)
sim_p.fill(np.nan)

for i_sim in range(n_sims):

    data = np.empty([n_per_group, n_groups])
    data.fill(np.nan)

    # simulate the data for this 'experiment'
    for i_group in range(n_groups):

        data[:, i_group] = np.random.normal(
            loc=group_means[i_group],
            scale=group_sigmas[i_group],
            size=n_per_group
        )

    result = scipy.stats.ttest_ind(data[:, 0], data[:, 1])

    sim_p[i_sim] = result[1]

# number of simulations where the null was rejected
n_rej = np.sum(sim_p < 0.05)

prop_rej = n_rej / float(n_sims)

print "Power: ", prop_rej
```

```
Power:  0.8527
```

We can see that our power to detect a large effect size with 30 participants per group in a between-subjects design is about 86%. That is, if a large effect size is truly present then we would expect to be able to reject the null hypothesis (at an alpha of 0.05) about 86% of the time.

The above code was written with clarity rather than speed of execution in mind, but in subsequent examples we would like to perform more computationally demanding analyses—so let's make some tweaks to allow it to be performed quicker. In the code below, we generate the simulated data all at once and then use the `axis` argument to `scipy.stats.ttest_ind`.

```python
import numpy as np

import scipy.stats

n_per_group = 30

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

# number of simulations
n_sims = 10000

data = np.empty([n_sims, n_per_group, n_groups])
```

```
data.fill(np.nan)

for i_group in range(n_groups):

    data[:, :, i_group] = np.random.normal(
        loc=group_means[i_group],
        scale=group_sigmas[i_group],
        size=[n_sims, n_per_group]
    )

result = scipy.stats.ttest_ind(
    data[:, :, 0],
    data[:, :, 1],
    axis=1
)

sim_p = result[1]

# number of simulations where the null was rejected
n_rej = np.sum(sim_p < 0.05)

prop_rej = n_rej / float(n_sims)

print "Power: ", prop_rej
```

```
Power:  0.8645
```

Required sample size to achieve a given power for a given effect size

Now we will move on to a more common scenario, where you have a design and effect size in mind and would like to know what sample size you would need to achieve a particular power (80% is typical). This is a straightforward extension of the previous example: we begin with a sample size and calculate the associated power. We then perform such a calculation repeatedly, each time increasing the sample size, until the power has reached the desired level.

```
import numpy as np

import scipy.stats

# start at 20 participants
n_per_group = 20

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

# number of simulations
n_sims = 10000

# power level that we would like to reach
desired_power = 0.8

# initialise the power for the current sample size to a small value
current_power = 0.0

# keep iterating until desired power is obtained
while current_power < desired_power:

    data = np.empty([n_sims, n_per_group, n_groups])
    data.fill(np.nan)

    for i_group in range(n_groups):

        data[:, :, i_group] = np.random.normal(
            loc=group_means[i_group],
            scale=group_sigmas[i_group],
            size=[n_sims, n_per_group]
        )

    result = scipy.stats.ttest_ind(
        data[:, :, 0],
        data[:, :, 1],
        axis=1
    )
    sim_p = result[1]

    # number of simulations where the null was rejected
    n_rej = np.sum(sim_p < 0.05)

    prop_rej = n_rej / float(n_sims)

    current_power = prop_rej

    print "With {n:d} samples per group, power = {p:.3f}".format(
        n=n_per_group,
        p=current_power
    )

    # increase the number of samples by one for the next iteration of the loop
    n_per_group += 1
```

```
With 20 samples per group, power = 0.690
With 21 samples per group, power = 0.713
With 22 samples per group, power = 0.743
With 23 samples per group, power = 0.750
With 24 samples per group, power = 0.777
With 25 samples per group, power = 0.793
With 26 samples per group, power = 0.813
```

We can see that we would reach the desired power with somewhere between 25 and 27 participants per group.

Visualising the sample size/power relationship

In the previous example, we sought to determine the sample size that would provide a given level of power. However, perhaps we do not have a single level of power in mind at the moment but would like to see the relationship between sample size and power so that we can see the costs and benefits of a particular sample size.

First, we will use a similar approach to the previous example, however we will perform the simulations across a fixed set of sample sizes. We will then save the power values to disk so that we can use them to create a visualisation without having to re-run all the simulations.

```
import numpy as np

import scipy.stats

# let's look at samples sizes of 10 per group up to 50 per group in steps of 5
ns_per_group = np.arange(10, 51, 5)

# a bit awkward - number of n's
n_ns = len(ns_per_group)

# effect size = 0.8
group_means = [0.0, 0.8]
group_sigmas = [1.0, 1.0]

n_groups = len(group_means)

# number of simulations
n_sims = 10000

power = np.empty(n_ns)
power.fill(np.nan)

for i_n in range(n_ns):

    n_per_group = ns_per_group[i_n]

    data = np.empty([n_sims, n_per_group, n_groups])
    data.fill(np.nan)

    for i_group in range(n_groups):

        data[:, :, i_group] = np.random.normal(
            loc=group_means[i_group],
            scale=group_sigmas[i_group],
            size=[n_sims, n_per_group]
        )

    result = scipy.stats.ttest_ind(
        data[:, :, 0],
        data[:, :, 1],
        axis=1
    )

    sim_p = result[1]

    # number of simulations where the null was rejected
    n_rej = np.sum(sim_p < 0.05)

    prop_rej = n_rej / float(n_sims)

    power[i_n] = prop_rej

# this is a bit tricky
# what we want to save is an array with 2 columns, but we have 2 different 1D
# arrays. here, we use the 'np.newaxis' property to add a column each of the
# two 1D arrays, and then 'stack' them horizontally
array_to_save = np.hstack(
    [
        ns_per_group[:, np.newaxis],
        power[:, np.newaxis]
    ]
)

power_path = "prog_data_vis_power_sim_data.tsv"

np.savetxt(
    power_path,
    array_to_save,
    delimiter="\t",
    header="Simulated power as a function of samples per group"
)
```

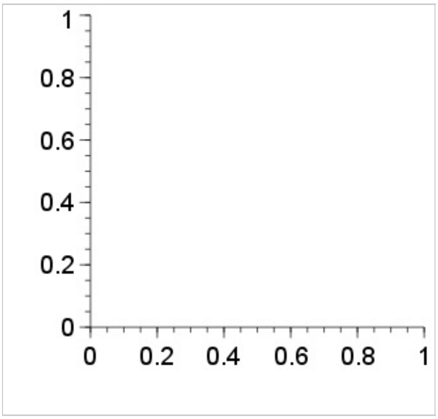
Now, we want to begin our visualisation code by loading the data we just saved.

```
import numpy as np

power_path = "prog_data_vis_power_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of n's is the number of rows
n_ns = power.shape[0]

ns_per_group = power[:, 0]
power_per_ns = power[:, 1]
```

We will now use familiar techniques to set up a one-panel figure. We will also apply some of the usual formatting changes.

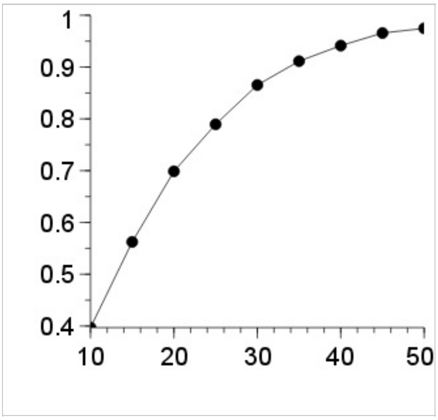
```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of n's is the number of rows
n_ns = power.shape[0]

ns_per_group = power[:, 0]
power_per_ns = power[:, 1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")


# do the typical manipulations to the axis apperances
graph.Border.hide.val = True
```

```
typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



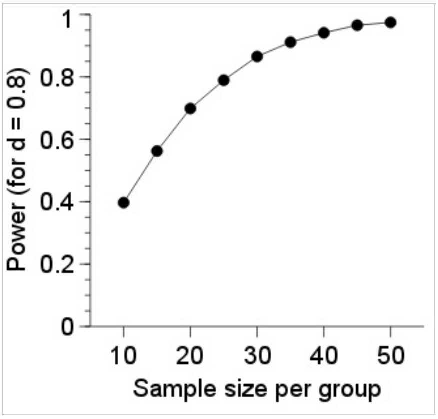
Alright, now we can plot the sample size per group on the horizontal axis and the power on the vertical axis. We will use the familiar `xy` figure type.

```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of n's is the number of rows
n_ns = power.shape[0]

ns_per_group = power[:, 0]
power_per_ns = power[:, 1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

# let veusz know about the data
embed.SetData("ns_per_group", ns_per_group)
embed.SetData("power_per_ns", power_per_ns)

xy = graph.Add("xy")

xy.xData.val = "ns_per_group"
xy.yData.val = "power_per_ns"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



The figure is looking pretty good—the decelerating relationship between the number of samples in each group and the power is clearly evident. Let's apply some more formatting changes to improve its appearance.

```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of n's is the number of rows
n_ns = power.shape[0]

ns_per_group = power[:, 0]
power_per_ns = power[:, 1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

# let veusz know about the data
embed.SetData("ns_per_group", ns_per_group)
embed.SetData("power_per_ns", power_per_ns)

xy = graph.Add("xy")

xy.xData.val = "ns_per_group"
xy.yData.val = "power_per_ns"

# set the x ticks to be at every 2nd location we sampled
x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"
x_axis.min.val = float(np.min(ns_per_group) - 5)
x_axis.max.val = float(np.max(ns_per_group) + 5)

# sensible to include the whole range here
y_axis.min.val = 0.0
y_axis.max.val = 1.0
y_axis.label.val = "Power (for d = 0.8)"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



Calculating power across varying sample and effect sizes

In the previous examples, we have assumed a fixed ('large') effect size. However, perhaps we want to investigate how power changes with both effect size and sample size. This is again a straightforward extension of the previous example. We will again save the data to disk to facilitate subsequent visualisation.

```python
import numpy as np

import scipy.stats

# let's look at samples sizes of 10 per group up to 50 per group in steps of 5
ns_per_group = np.arange(10, 51, 5)

# a bit awkward - number of n's
n_ns = len(ns_per_group)

# span the range from a bit less than a 'small' effect size to a bit bigger
# than a 'large' effect size
effect_sizes = np.arange(0.2, 0.91, 0.1)

n_effect_sizes = len(effect_sizes)

power = np.empty([n_effect_sizes, n_ns])
power.fill(np.nan)

# number of simulations
n_sims = 10000

for i_es in range(n_effect_sizes):

    group_means = [0.0, effect_sizes[i_es]]
    group_sigmas = [1.0, 1.0]

    n_groups = len(group_means)

    for i_n in range(n_ns):

        n_per_group = ns_per_group[i_n]

        data = np.empty([n_sims, n_per_group, n_groups])
        data.fill(np.nan)

        for i_group in range(n_groups):

            data[:, :, i_group] = np.random.normal(
                loc=group_means[i_group],
                scale=group_sigmas[i_group],
                size=[n_sims, n_per_group]
            )

        result = scipy.stats.ttest_ind(
            data[:, :, 0],
            data[:, :, 1],
            axis=1
        )

        sim_p = result[1]

        # number of simulations where the null was rejected
        n_rej = np.sum(sim_p < 0.05)

        prop_rej = n_rej / float(n_sims)

        power[i_es, i_n] = prop_rej

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

np.savetxt(
    power_path,
    power,
    delimiter="\t",
    header="Simulated power as a function of samples per group and ES"
)
```

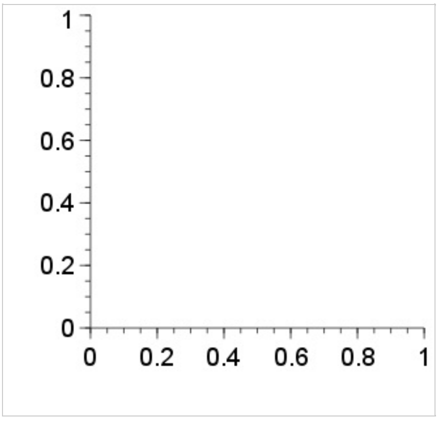
Visualising power across varying sample and effect sizes

Now that we have our power calculations as a function of sample size and effect size saved to disk, we can think about how to visualise it. But first, let's load the data.

```
import numpy as np

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)
```

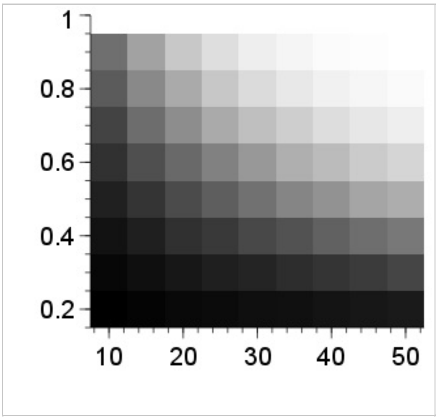
To visualise this two-dimensional data (effect size x sample size per group), we will investigate a new veusz figure type—an `image`. First, let's set up our figure framework as per usual.

```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:
    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



Now, we will add our `image` figure type to the graph. First, we tell veusz about the data using the `SetData2D` function, using the arguments `xcent` and `ycent` to tell veusz what the horizontal and vertical datapoints represent.

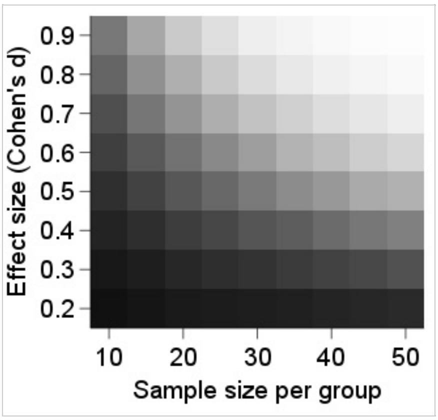
```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData2D(
    "power",
    power,
    xcent=ns_per_group,
    ycent=effect_sizes
)

img = graph.Add("image")

img.data.val = "power"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



Now we are starting to get somewhere—we can see that the power for a given combination of effect size and sample size per group is represented by the luminance of the relevant cell. Let's first clean up the appearance of the figure a bit.

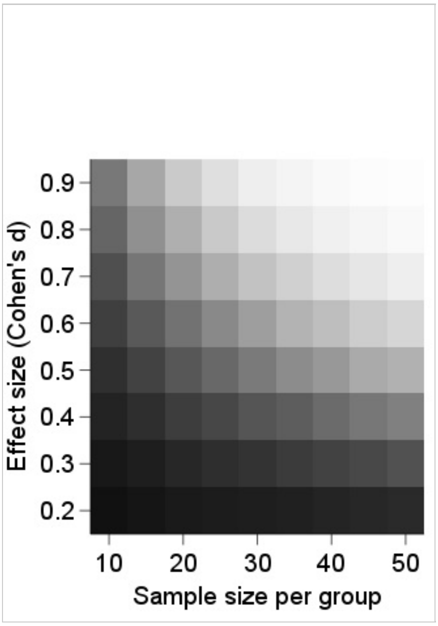
```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData2D(
    "power",
    power,
    xcent=ns_per_group,
    ycent=effect_sizes
)

img = graph.Add("image")

img.data.val = "power"

# set 0.0 = black and 1.0 = white
img.min.val = 0.0
img.max.val = 1.0

x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"

y_axis.MajorTicks.manualTicks.val = effect_sizes.tolist()
y_axis.MinorTicks.hide.val = True
y_axis.label.val = "Effect size (Cohen's d)"
y_axis.max.val = float(np.max(effect_sizes) + 0.05)

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True
```

```
typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



It is looking better, but a major deficiency at the moment is a lack of information about what the intensity in each cell represents. To remedy this, we will add a `colorbar` to our graph. But first, we will make room for it by increasing our vertical page size and increasing the top margin of our graph.

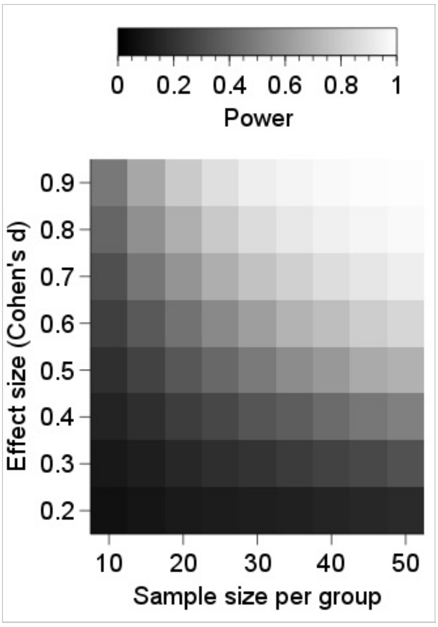
```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "12cm"

graph = page.Add("graph", autoadd=False)

graph.topMargin.val = "3cm"

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData2D(
    "power",
    power,
    xcent=ns_per_group,
    ycent=effect_sizes
)

img = graph.Add("image")

img.data.val = "power"

# set 0.0 = black and 1.0 = white
img.min.val = 0.0
img.max.val = 1.0

x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"

y_axis.MajorTicks.manualTicks.val = effect_sizes.tolist()
y_axis.MinorTicks.hide.val = True
y_axis.label.val = "Effect size (Cohen's d)"
y_axis.max.val = float(np.max(effect_sizes) + 0.05)

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



Now, we can add and position the colour bar:

```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "12cm"

graph = page.Add("graph", autoadd=False)

graph.topMargin.val = "3cm"

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")
```

```
embed.SetData2D(
    "power",
    power,
    xcent=ns_per_group,
    ycent=effect_sizes
)

# give the image a 'name', so that we can tell the colour bar what image it
# should be associated with
img = graph.Add("image", name="power_img")

img.data.val = "power"

# set 0.0 = black and 1.0 = white
img.min.val = 0.0
img.max.val = 1.0

cbar = graph.Add("colorbar")

# connect the colour bar with the power image
cbar.widgetName.val = "power_img"
cbar.horzPosn.val = "centre"
cbar.vertPosn.val = "manual"
cbar.vertManual.val = -0.35
cbar.label.val = "Power"

x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"

y_axis.MajorTicks.manualTicks.val = effect_sizes.tolist()
y_axis.MinorTicks.hide.val = True
y_axis.label.val = "Effect size (Cohen's d)"
y_axis.max.val = float(np.max(effect_sizes) + 0.05)

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis, cbar]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```



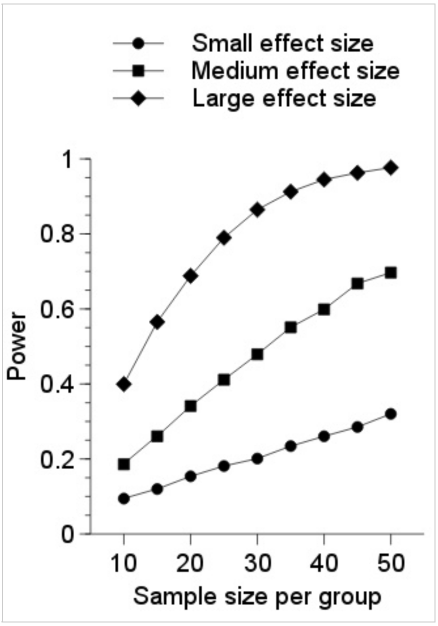
Exercises

1. What point does the following code make in relation to the previous example where we calculated the power of an independent-samples t-test?

```
import numpy as np

import scipy.stats

n = 30

# effect size = 0.8
cond_means = [0.0, 0.8]
n_cond = len(cond_means)

cov = [[1, 0.5], [0.5, 1]]

# number of simulations
n_sims = 10000

data = np.random.multivariate_normal(
    cond_means,
    cov,
    size=[n_sims, n]
)

result = scipy.stats.ttest_rel(
    data[:, :, 0],
    data[:, :, 1],
    axis=1
)

sim_p = result[1]

# number of simulations where the null was rejected
n_rej = np.sum(sim_p < 0.05)

prop_rej = n_rej / float(n_sims)

print "Power: ", prop_rej
```

```
Power:  0.9895
```

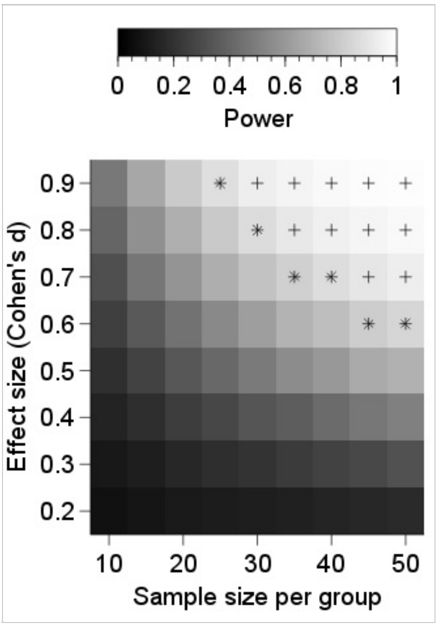
2. In the last example, we used an image to display the relationship between effect size, sample size per group, and power. Create an alternative visualisation use a series of line plots depict the relationship between sample size per group and power separately for three effect sizes (as shown below).



Hint: the following with add the legend to the figure, and set the `key.val` property of each `xy`:

```
key = graph.Add("key")

key.horzPosn.val = "centre"
key.vertPosn.val = "manual"
key.vertManual.val = 1.125
key.Text.font.val = "Arial"
key.Border.hide.val = True
```

3. In the last example, it is difficult to determine the precise power of each cell. Add elements to the figure so that any cell with a power between 0.8 and 0.9 is marked with an "asterisk" and any cell with a power greater than 0.9 is marked with a "lineplus" (hint: set `marker.val`, and create the markers before the image so they are visible).



Back to top

Damien J. Mannion, Ph.D. — School of Psychology — UNSW Australia

Programming for Psychology in Python

Data Analysis and Visualisation

Exercise solutions

Scroll down to see the solutions to the exercises. The different lessons have quite large spacing so that you don't accidentally see the solutions for other lessons.

Arrays

1.

```
import numpy as np

# array from list = 1D array
data = np.array([1, 2, 3, 4])
print data

# array from list of lists = 2D array
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print data
```

```
[1 2 3 4]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

2.

```
import numpy as np

data = np.random.random(10)

print data

in_interval = np.logical_and(
    data >= 0.2,
    data <= 0.6
)

print data[in_interval]
```

```
[ 0.19296709  0.61385947  0.74659068  0.31168346  0.86466776  0.52512645
  0.58529987  0.92367176  0.733665    0.26392719]
[ 0.31168346  0.52512645  0.58529987  0.26392719]
```

3.

```
import numpy as np

# generate a 3D array
data = np.random.random([2, 3, 4])

print data

summed_array = np.sum(np.sum(data, axis=0), axis=-1)

print summed_array
```

```
[[[ 0.05500094  0.41588443  0.65494624  0.8270507 ]
  [ 0.13430731  0.01110865  0.28208219  0.01980629]
  [ 0.79463058  0.10939312  0.36727225  0.37713932]]

 [[ 0.11469447  0.80578551  0.36093281  0.97665116]
  [ 0.65142678  0.13242163  0.59296513  0.47941308]
  [ 0.13195652  0.35096913  0.00983718  0.41312592]]]
[ 4.21094626  2.30353107  2.55432403]
```

4.

```
import numpy as np

data = np.random.random([4, 3])

np.savetxt(
    "data.txt",
    data,
    header="This header string would contain useful info"
)
```

Creating figures

1. The `bar.barfill.val` value controls the 'width' of each bar. The widths are allowed to vary between 0 and 1 - anything outside this range generates an error.
2. The higher DPI image in Word looks a lot nicer, with appreciably higher resolution. Viewing the PDF in Acrobat shows that one can zoom a long way in and still have very high resolution. Trying the same thing on the images in Word shows that the high DPI is able to tolerate zooming much better than the low DPI image.
3.

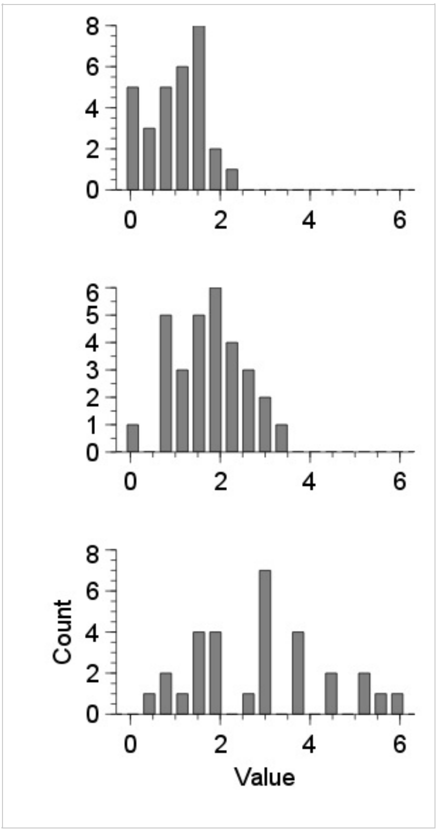
```python
import numpy as np

import veusz.embed

n_samples = 1000

# generate random samples from a Gaussian (normal) distribution
data = np.random.normal(loc=0.0, scale=1.0, size=n_samples)

# we were told to use 15 bins between -3 and +3
n_bins = 15
bin_range = [-3.0, +3.0]

# using CTRL-i on np.histogram gives a lot of useful information
[hist, bin_edges] = np.histogram(a=data, bins=n_bins, range=bin_range)

# we have the bin edges, but ideally for the figure we will have the
# bin centres - so let's work them out

# work out the size of the bins (length between the edges)
bin_size = bin_edges[1] - bin_edges[0]

# now add it to the edges
bin_centres = bin_edges + bin_size

# right, now we can start drawing our figure
# below is mostly as per the Poisson example

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "6cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

bar = graph.Add("bar")

embed.SetData("bins", bin_centres)
embed.SetData("bin_counts", hist)

bar.posn.val = "bins"
bar.lengths.val = "bin_counts"

x_axis.label.val = "Value"
y_axis.label.val = "Count"

# calculate the minimum
x_axis.min.val = float(np.min(bin_centres) - 0.5)
x_axis.max.val = float(np.max(bin_centres) + 0.5)
x_axis.MinorTicks.hide.val = True

# these tick marks look pretty ugly - lets make our own
#x_axis.MajorTicks.manualTicks.val = bin_centres.tolist()
x_axis.MajorTicks.manualTicks.val = range(-3, 4)

x_axis.autoMirror.val = False
x_axis.outerticks.val = True

y_axis.MinorTicks.hide.val = True
y_axis.autoMirror.val = False
y_axis.outerticks.val = True

graph.Border.hide.val = True

typeface = "Arial"
```

```python

for curr_axis in [x_axis, y_axis]:
    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

embed.WaitForClose()
```

Descriptive statistics—calculation and visualisation

1.

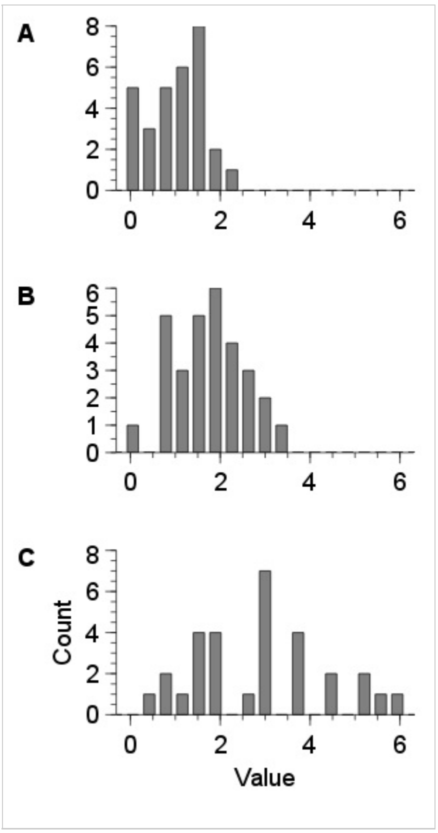
```python
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = n_cond
grid.columns.val = 1

grid.leftMargin.val = "0.5cm"
grid.bottomMargin.val = "0.5cm"

min_val = np.min(data)
max_val = np.max(data)

bin_pad = 0.5

n_bins = 20

bin_edges = np.linspace(
    min_val - bin_pad,
    max_val + bin_pad,
    n_bins,
    endpoint=True
)

bin_delta = bin_edges[1] - bin_edges[0]

bin_centres = bin_edges + (bin_delta / 2.0)

embed.SetData("bin_centres", bin_centres[:-1])

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

    hist_result = np.histogram(data[:, i_cond], bin_edges)

    bin_counts = hist_result[0]

    bin_counts_str = "bin_counts_{i:d}".format(i=i_cond)

    embed.SetData(bin_counts_str, bin_counts)

    bar = graph.Add("bar")

    bar.posn.val = "bin_centres"
    bar.lengths.val = bin_counts_str

    if i_cond == (n_cond - 1):
        x_axis.label.val = "Value"
        y_axis.label.val = "Count"

    graph.Border.hide.val = True

    typeface = "Arial"

    for curr_axis in [x_axis, y_axis]:
        curr_axis.Label.font.val = typeface
        curr_axis.TickLabels.font.val = typeface

        curr_axis.autoMirror.val = False
        curr_axis.outerticks.val = True

embed.WaitForClose()
```



2.

```
import numpy as np

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "16cm"

grid = page.Add("grid")

grid.rows.val = n_cond
grid.columns.val = 1

grid.leftMargin.val = "0.5cm"
grid.bottomMargin.val = "0.5cm"

min_val = np.min(data)
max_val = np.max(data)

bin_pad = 0.5

n_bins = 20

bin_edges = np.linspace(
    min_val - bin_pad,
```

```python
    max_val + bin_pad,
    n_bins,
    endpoint=True
)

bin_delta = bin_edges[1] - bin_edges[0]

bin_centres = bin_edges + (bin_delta / 2.0)

embed.SetData("bin_centres", bin_centres[:-1])

panel_labels = ["A", "B", "C"]

for i_cond in range(n_cond):

    graph = grid.Add("graph", autoadd=False)

    x_axis = graph.Add("axis")
    y_axis = graph.Add("axis")

    hist_result = np.histogram(data[:, i_cond], bin_edges)

    bin_counts = hist_result[0]

    bin_counts_str = "bin_counts_{i:d}".format(i=i_cond)

    embed.SetData(bin_counts_str, bin_counts)

    bar = graph.Add("bar")

    bar.posn.val = "bin_centres"
    bar.lengths.val = bin_counts_str

    if i_cond == (n_cond - 1):
        x_axis.label.val = "Value"
        y_axis.label.val = "Count"

    graph.Border.hide.val = True

    typeface = "Arial"

    for curr_axis in [x_axis, y_axis]:

        curr_axis.Label.font.val = typeface
        curr_axis.TickLabels.font.val = typeface

        curr_axis.autoMirror.val = False
        curr_axis.outerticks.val = True

    label = graph.Add("label")
    label.label.val = panel_labels[i_cond]
    label.xPos.val = -0.33
    label.yPos.val = 0.9
    label.Text.bold.val = True
    label.Text.font.val = typeface

embed.WaitForClose()
```



3.

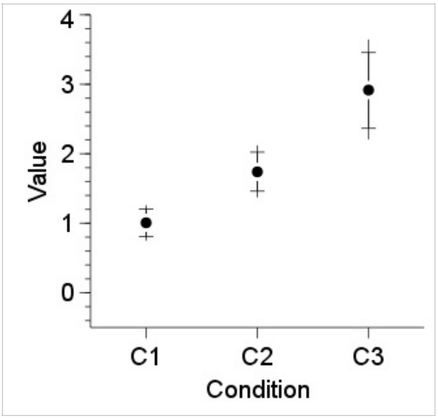
```python
import numpy as np
import scipy.stats

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# only looking at the first condition
data = data[:, 0]

data_mean = np.mean(data)
data_median = np.median(data)

# 25th and 75th percentiles
p25 = scipy.stats.scoreatpercentile(data, 25)
p75 = scipy.stats.scoreatpercentile(data, 75)

# inter-quartile range
iqr = p75 - p25

bottom_whisker = p25 - 1.5 * iqr
top_whisker = p75 + 1.5 * iqr

i_outliers = np.logical_or(
    data < (p25 - 1.5 * iqr),
    data > (p75 + 1.5 * iqr)
)

outliers = data[i_outliers]

print "Mean: ", data_mean
print "Median: ", data_median
print "25th percentile: ", p25
print "75th percentile: ", p75
print "IQR: ", iqr
print "Top whisker: ", top_whisker
print "Bottom whisker: ", bottom_whisker
print "Outliers: ", outliers
```

```
Mean:  1.00667777645
Median:  1.05751833665
25th percentile:  0.567579127024
75th percentile:  1.42671256929
IQR:  0.85913344227
```

```
Top whisker:  2.7154127327
Bottom whisker:  -0.721121036382
Outliers:  []
```

4. This might not be the best solution, with the duplication somewhat awkward—seems to work though.

```python
import numpy as np
import scipy.stats

import veusz.embed

data_path = "prog_data_vis_descriptives_sim_data.tsv"

# load the data we generated in the previous step
data = np.loadtxt(data_path, delimiter="\t")

# pull out some design parameters from the shape of the array
n_per_group = data.shape[0]
n_cond = data.shape[1]
subjects = range(n_per_group)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

n_boot = 10000

cond_labels = ["C1", "C2", "C3"]

cis = [95, 99]
ci_styles = ["barends", "bar"]

for i_cond in range(n_cond):

    cond_mean = np.mean(data[:, i_cond])

    # do bootstrapping
    boot_data = np.empty((n_boot))
    boot_data.fill(np.nan)

    for i_boot in range(n_boot):

        rand_subj_sample = np.random.choice(
            a=subjects,
            replace=True,
            size=n_per_group
        )

        boot_data[i_boot] = np.mean(data[rand_subj_sample, i_cond])

    for i_ci_type in xrange(2):

        ci = cis[i_ci_type]

        lower_ci = scipy.stats.scoreatpercentile(
            boot_data,
            (100 - ci) / 2.0
        )
        upper_ci = scipy.stats.scoreatpercentile(
            boot_data,
            100 - (100 - ci) / 2.0
        )

        pos_err = upper_ci - cond_mean
        neg_err = lower_ci - cond_mean

        data_str = "cond_{i:d}_{ic:d}".format(i=i_cond, ic=i_ci_type)

        # cond_mean is a single value whereas SetData is expecting a list or
        # array, so pass a one-item list
        embed.SetData(
            data_str,
            [cond_mean],
            poserr=[pos_err],
            negerr=[neg_err]
        )

        xy = graph.Add("xy")

        xy.xData.val = i_cond
        xy.yData.val = data_str
        xy.labels.val = cond_labels[i_cond]

        xy.Label.hide.val = True
        xy.MarkerLine.color.val = "white"
        xy.MarkerLine.width.val = "2pt"

        xy.markerSize.val = "4pt"

        xy.errorStyle.val = ci_styles[i_ci_type]

x_axis.mode.val = "labels"
x_axis.MajorTicks.manualTicks.val = range(n_cond)
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Condition"
x_axis.min.val = -0.5
x_axis.max.val = n_cond - 1 + 0.5

y_axis.min.val = -0.5
y_axis.max.val = 4.0
y_axis.label.val = "Value"

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()

embed.Export("descrip_4.png", backcolor="white")
```



Inferential statistics and scatter plots

1. The false positive rate roughly follows 0.05 * k, where k is the number of tests. For example, with 5 tests:

```python
import numpy as np

import scipy.stats

n = 30

pop_mean = 0.0
pop_sigma = 1.0

# number of conditions
n_conds = 5

# number of simulations we will run
n_sims = 10000

# store the p value associated with each simulation
p_values = np.empty([n_sims, n_conds])
p_values.fill(np.nan)

for i_sim in range(n_sims):

    # perform our 'experiment'
    data = np.random.normal(
        loc=pop_mean,
        scale=pop_sigma,
        size=[n, n_conds]
    )

    for i_cond in range(n_conds):

        # perform the t-test
        result = scipy.stats.ttest_1samp(
            a=data[:, i_cond],
            popmean=0.0
        )

        # save the p value
        p_values[i_sim, i_cond] = result[1]

assert np.sum(np.isnan(p_values)) == 0

# determine whether, on each simulation, either of the two tests were deemed to
# be significantly different from 0
either_rej = np.any(p_values < 0.05, axis=1)

n_null_rej = np.sum(either_rej)

# proportion of null hypotheses rejected
prop_null_rej = n_null_rej / float(n_sims)

print prop_null_rej
```

```
0.2282
```

2.

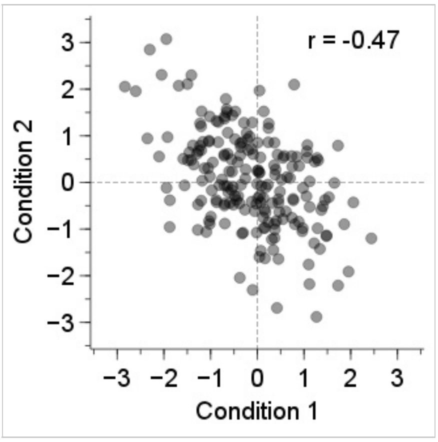
```python
import numpy as np

import scipy.stats

n = 30

pop_mean = 0.0
pop_sigma = 1.0

# number of conditions
n_conds = 2

# number of simulations we will run
n_sims = 10000

# store the p value associated with each simulation
p_values = np.empty([n_sims, n_conds])
p_values.fill(np.nan)
```

```
for i_sim in range(n_sims):

    # perform our 'experiment'
    data = np.random.normal(
        loc=pop_mean,
        scale=pop_sigma,
        size=[n, n_conds]
    )

    for i_cond in range(n_conds):

        # perform the t-test
        result = scipy.stats.ttest_1samp(
            a=data[:, i_cond],
            popmean=0.0
        )

        # save the p value
        p_values[i_sim, i_cond] = result[1]

assert np.sum(np.isnan(p_values)) == 0

# determine whether, on each simulation, either of the two tests were deemed to
# be significantly different from 0

critical_p = 0.05 / n_conds

either_rej = np.any(p_values < critical_p, axis=1)

n_null_rej = np.sum(either_rej)

# proportion of null hypotheses rejected
prop_null_rej = n_null_rej / float(n_sims)

print prop_null_rej
```

```
0.0505
```

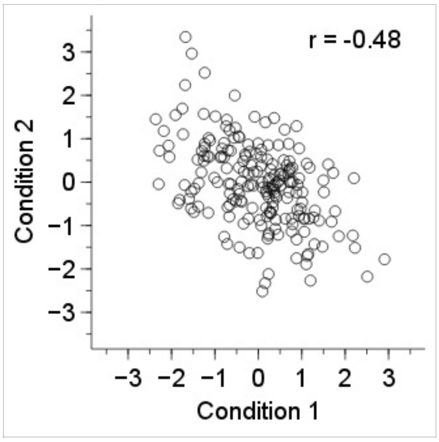
3.

```
import numpy as np

import scipy.stats

import veusz.embed

n = 200

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True
xy.MarkerFill.transparency.val = 60
xy.MarkerLine.transparency.val = 60


graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

x_grid = graph.Add("xy")
# for simple data such as this, we can specify it directly rather than using
# embed.SetData
x_grid.xData.val = [-axis_extreme, axis_extreme]
x_grid.yData.val = [0.0, 0.0]

y_grid = graph.Add("xy")
y_grid.yData.val = [-axis_extreme, axis_extreme]
y_grid.xData.val = [0.0, 0.0]

for grid in [x_grid, y_grid]:
    grid.PlotLine.style.val = "dashed"
    grid.PlotLine.color.val = "grey"
    grid.MarkerFill.hide.val = True
    grid.MarkerLine.hide.val = True

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

result = scipy.stats.pearsonr(x=data[:, 0], y=data[:, 1])
r = result[0]

r_fig = graph.Add("label")

r_fig.label.val = "r = {r:.2f}".format(r=r)
r_fig.xPos.val = 0.65
r_fig.yPos.val = 0.9
r_fig.Text.font.val = typeface

embed.WaitForClose()

embed.Export("inferential_3.png", backcolor="white")
```



4.

```
import numpy as np
import scipy.stats

import veusz.embed

n = 200

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "8.4cm"

graph = page.Add("graph", autoadd=False)

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("x_data", data[:, 0])
embed.SetData("y_data", data[:, 1])

xy = graph.Add("xy")

xy.xData.val = "x_data"
xy.yData.val = "y_data"

xy.PlotLine.hide.val = True

xy.MarkerFill.hide.val = True

graph.aspect.val = 1.0

axis_pad = 0.5

axis_extreme = np.max(np.abs(data)) + axis_pad

graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.min.val = float(-axis_extreme)
    curr_axis.max.val = float(+axis_extreme)

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

x_axis.label.val = "Condition 1"
y_axis.label.val = "Condition 2"

result = scipy.stats.pearsonr(x=data[:, 0], y=data[:, 1])
r = result[0]

r_fig = graph.Add("label")

r_fig.label.val = "r = {r:.2f}".format(r=r)
r_fig.xPos.val = 0.65
r_fig.yPos.val = 0.9
r_fig.Text.font.val = typeface

embed.WaitForClose()

embed.Export("inferential_4.png", backcolor="white")
```



5.

```
import numpy as np

import scipy.stats

n = 30

true_corr = -0.5

# don't worry too much about this - just a way to generate random numbers with
# the desired correlation
cov = [[1.0, true_corr], [true_corr, 1.0]]

data = np.random.multivariate_normal(
    mean=[0.0, 0.0],
    cov=cov,
    size=n
)

result = scipy.stats.pearsonr(x=data[:, 0], y=data[:, 1])
data_r = result[0]

n_boot = 10000

boot_data = np.empty(n_boot)
boot_data.fill(np.nan)

for i_boot in range(n_boot):

    rand_subj_sample = np.random.choice(
        a=range(n),
        replace=True,
        size=n
    )

    result = scipy.stats.pearsonr(
        x=data[rand_subj_sample, 0],
        y=data[rand_subj_sample, 1]
    )

    boot_data[i_boot] = result[0]

lower_ci = scipy.stats.scoreatpercentile(boot_data, 2.5)
upper_ci = scipy.stats.scoreatpercentile(boot_data, 97.5)

print "Bootstrapped 95% CI: [{a:.2f}, {b:.2f}]".format(
    a=lower_ci, b=upper_ci
)

lower_par_ci = np.tanh(np.arctanh(data_r) - 1.96 * 0.19245)
upper_par_ci = np.tanh(np.arctanh(data_r) + 1.96 * 0.19245)

print "Parametric 95% CI: [{a:.2f}, {b:.2f}]".format(
    a=lower_par_ci, b=upper_par_ci
)
```

```
Bootstrapped 95% CI: [-0.57, -0.03]
Parametric 95% CI: [-0.61, 0.04]
```

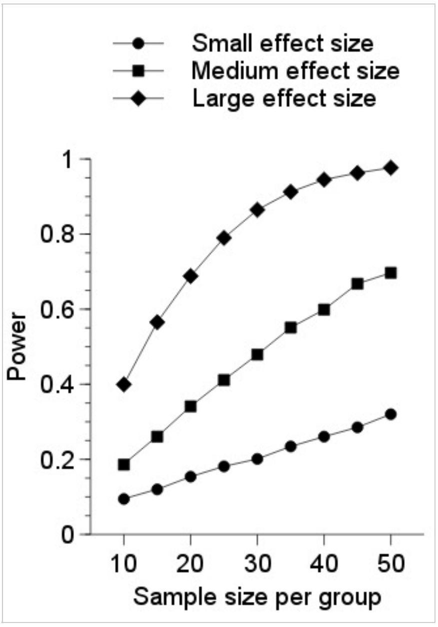
Experimental design—power analysis and its visualisation

1. This analysis is showing the power for a within-subjects design. Despite having half the total number of participants (n = 30, as opposed to n = 30 per group for two groups), the power to detect the large effect size is substantially higher for this within-subjects design.

2.

```python
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "12cm"

graph = page.Add("graph", autoadd=False)

graph.topMargin.val = "3cm"

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")

embed.SetData("ns_per_group", ns_per_group)

# hardcode these; would be better to use a function to work them out
i_es_to_plot = [1, 3, 6]

n_es_to_plot = len(i_es_to_plot)

es_labels = [
    "Small effect size",
    "Medium effect size",
    "Large effect size"
]
es_markers = ["circle", "square", "diamond"]

n_es_to_plot = len(i_es_to_plot)

for i_es in range(n_es_to_plot):

    es_str = "power_{es:d}".format(es=i_es)

    embed.SetData(es_str, power[i_es_to_plot[i_es], :])

    xy = graph.Add("xy")

    xy.xData.val = "ns_per_group"
    xy.yData.val = es_str

    xy.key.val = es_labels[i_es]

    xy.marker.val = es_markers[i_es]

# set the x ticks to be at every 2nd location we sampled
x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"
x_axis.min.val = float(np.min(ns_per_group) - 5)
x_axis.max.val = float(np.max(ns_per_group) + 5)

# sensible to include the whole range here
y_axis.min.val = 0.0
y_axis.max.val = 1.0
y_axis.label.val = "Power"
```

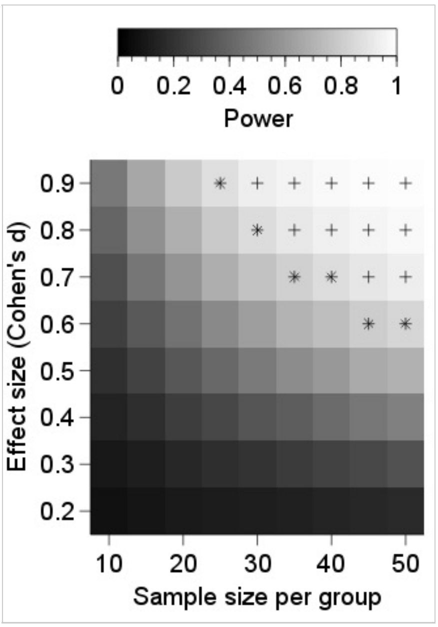
```
# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

key = graph.Add("key")

key.horzPosn.val = "centre"
key.vertPosn.val = "manual"
key.vertManual.val = 1.125
key.Text.font.val = "Arial"
key.Border.hide.val = True

embed.WaitForClose()
```



3.

```
import numpy as np

import veusz.embed

power_path = "prog_data_vis_power_es_x_ss_sim_data.tsv"

power = np.loadtxt(power_path, delimiter="\t")

# number of effect sizes is the number of rows
n_effect_sizes = power.shape[0]
# number of n's is the number of columns
n_ns = power.shape[1]

ns_per_group = np.arange(10, 51, 5)
effect_sizes = np.arange(0.2, 0.91, 0.1)

embed = veusz.embed.Embedded("veusz")

page = embed.Root.Add("page")
page.width.val = "8.4cm"
page.height.val = "12cm"

graph = page.Add("graph", autoadd=False)

graph.topMargin.val = "3cm"

x_axis = graph.Add("axis")
y_axis = graph.Add("axis")
```

```python
for i_n in range(n_ns):
    for i_es in range(n_effect_sizes):

        if power[i_es, i_n] > 0.8:

            if power[i_es, i_n] > 0.9:
                marker = "lineplus"
            else:
                marker = "asterisk"

            xy = graph.Add("xy")

            xy.xData.val = [ns_per_group[i_n]]
            xy.yData.val = [effect_sizes[i_es]]

            xy.marker.val = marker

embed.SetData2D(
    "power",
    power,
    xcent=ns_per_group,
    ycent=effect_sizes
)

# give the image a 'name', so that we can tell the colour bar what image it
# should be associated with
img = graph.Add("image", name="power_img")

img.data.val = "power"

# set 0.0 = black and 1.0 = white
img.min.val = 0.0
img.max.val = 1.0

cbar = graph.Add("colorbar")

# connect the colour bar with the power image
cbar.widgetName.val = "power_img"
cbar.horzPosn.val = "centre"
cbar.vertPosn.val = "manual"
cbar.vertManual.val = -0.35
cbar.label.val = "Power"

x_axis.MajorTicks.manualTicks.val = ns_per_group[::2].tolist()
x_axis.MinorTicks.hide.val = True
x_axis.label.val = "Sample size per group"

y_axis.MajorTicks.manualTicks.val = effect_sizes.tolist()
y_axis.MinorTicks.hide.val = True
y_axis.label.val = "Effect size (Cohen's d)"
y_axis.max.val = float(np.max(effect_sizes) + 0.05)

# do the typical manipulations to the axis apperances
graph.Border.hide.val = True

typeface = "Arial"

for curr_axis in [x_axis, y_axis, cbar]:

    curr_axis.Label.font.val = typeface
    curr_axis.TickLabels.font.val = typeface

    curr_axis.autoMirror.val = False
    curr_axis.outerticks.val = True

embed.WaitForClose()
```


0
0.2
0.4
0.6
0.8
1
Power
Effect size (Cohen's d)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
10
20
30
40
50
Sample size per group
