Lab 08: Mathematical Packages

Contents

1 Mathematical Operations **1**
1.1 `math` . . . 2
1.2 `random` . . . 2
1.3 `numpy` . . . 3

2 Statistics in `scipy.stats` **3**
2.1 Expected value . . . 3
2.2 Correlation . . . 4

3 Graphing with `matplotlib` **5**
3.1 Basic plots . . . 5
3.2 Scatter plots . . . 7
3.3 Bar graphs . . . 8

4 Putting It All Together **9**

5 Handing In **10**

Objectives

By the end of this lab you will understand:

- When to use which scientific computing modules
- The different types of graphs you can create

By the end of this lab you will be able to:

- Perform mathematical operations in Python
- Perform statistical tests using `scipy.stats`
- Create graphs using `matplotlib`

Instructions

As with Lab 05, this handout contains a lot of information. Some of it is new and may serve as a useful reference later on, and some of it is intended to summarize the statistical tests you learned in previous modules. You thus may find it best to start by reading solely the tasks, rather than all the material in order, and then going back and looking at the relevant sections only if/when you find it useful.

1 Mathematical Operations

You have already seen how to perform basic mathematical operations in Python, as most of these are built in. For example, you can perform numeric operations using `+`, `-`, `*`, `/`, `//`, `%`, etc., and create sequences of numbers using `range`.

Depending on how you use Python, it is possible that in the future you may end up wanting to do more than these basic operations. It is also very possible that you won't. Nevertheless, this section serves as a reference guide in the event that you do end up needing these operations (for a more thorough guide there is extensive documentation for all of these modules that you can find online).

1.1 `math`

The `math` module is most useful with respect to the `float` type, as the way this type is handled can cause frequent problems (both in Python and in most programming languages). For example, consider the following calculations:

```
>>> 10 - 9.99
0.009999999999999787
>>> 100 - 99.99
0.010000000000005116
>>> 10 - 9.99 == 100 - 99.99
False
```

This might seem a bit shocking: of course both of these simple subtractions are equal to 0.01, and yet Python doesn't represent either of them as exactly this value, and furthermore doesn't even think they're equal! While this is indeed quite weird, the topic of how to handle *floating-point arithmetic* in a programming language is complex and goes far beyond what we'll cover in this module. Instead, the best we can do is simply avoid `float` whenever possible. If you do need to use it, however, or need to use constants such as e and π, the `math` module may help you. Examples, assuming you've run `import math`, are:

- `math.pi` returns the mathematical constant π. Among many other potential uses, this is helpful in computing the area of a circle or in analyzing certain statistical distributions.
- `math.e` returns Euler's identity e. Again, this has many potential uses within mathematics, economics, probability, and statistics.
- `math.ceil(x)` returns the *ceiling* of a float x; i.e., the smallest integer greater than or equal to x. For example, `math.ceil(4.8)` returns 5.0.
- `math.floor(x)` returns the *floor* of a float x; i.e., the largest integer less than or equal to x. For example, `math.ceil(4.8)` returns 4.0.
- `math.log(x, b)` returns the logarithm of x to the base b. If b is not provided, this returns the *natural* logarithm; i.e., the logarithm to base e. For example, `math.log(32,2)` returns 5.0, since $2^5 = 32$.
- `math.sqrt(x)` returns the square root of x. For example, `math.sqrt(64)` returns 8.0.

In addition, there is one related built-in function that may be helpful; this is:

- `round(x, n)` returns the number x rounded to n decimal places. If n is not provided, this returns x rounded to the nearest integer value. For example, `round(5.26, 1)` returns 5.3.

1.2 `random`

The appropriately named `random` module can help you perform random samples, which may be helpful in statistical analyses and modeling. Some useful functions, assuming you've run `import random`, are:

- `random.randint(x, y)` returns a random integer between x and y. For example, `random.randint(0,10)` could return 8 (but should return every value between 0 and 10 with equal probability).
- `random.choice(space)` returns an element randomly sampled from `space`. For example, `random.choice(range(10))` returns the same thing as `random.randint(0,10)`.
- `random.sample(space, k)` returns a list of k elements randomly sampled from `space`.
- `random.gauss(mu, sig)` returns a randomly sampled element from the Gaussian (i.e., normal) distribution parameterized by `mu` and `sig`. Variants of this function exist for most other probability distributions.

Task 1. Create a file `random_graph.py`. In this file, add some code to create two random samples `x` and `y` of size 100 of numbers between 0 and 500.

Solution 1. The code for this (and subsequent tasks involving this file) is in `random_graph.py`. The lines for sampling are:

```
x = random.sample(range(500), 100)
y = random.sample(range(500), 100)
```

1.3 `numpy`

The main benefit of the NumPy (Numerical Python) package is when we want to handle matrices or *arrays* of objects. Without it, the only way to represent arrays in Python is using lists of lists, which is okay to do every once in a while but might quickly get tedious if you're doing it a lot.

NumPy is a powerful package with many features, so we will not attempt to cover it in any detail here. Instead, some particular functions that might be helpful, assuming you've run `import numpy as np` (the suggested abbreviation), are:

- `np.array(list_of_lists)` constructs an array given a list of lists representing a matrix. For example, `np.array([1,1],[2,2]])` returns

  ```
  array([[1, 1],
         [2, 2]])
  ```

- `np.arange(x,y,step)` returns the range beginning at `x`, ending at `y`, and increasing each time by `step`; it thus has the same functionality as `range` but can handle non-integer values. For example, `np.arange(0, 0.5, 0.1)` returns `array([0, 0.1, 0.2, 0.3, 0.4])` (which can be converted to a regular Python list using `list`).

This package also contains several modules that emulate the features of other modules, so if you already are importing it you may not need to import others explicitly. For example, `np.random` has all the same functions as the `random` module covered earlier (although, in case you're curious, it is actually a distinct implementation).

2 Statistics in `scipy.stats`

You have already seen many ways to perform statistics in Python last year, so this section should hopefully serve to refresh your memory of some of what you learned rather than introduce new things. Similarly, it won't cover the range of modules that you saw before: `pandas`, `statsmodel`, etc. Instead, we'll focus just on SciPy, which is another rich package intended for scientific computing. Again, rather than attempt to enumerate all its features, we'll focus on a single module, which is `scipy.stats`; this is typically imported using `import scipy.stats as ss`. A complete list of all of the features of this module can be found here: `https://docs.scipy.org/doc/scipy/reference/stats.html`.

2.1 Expected value

One of the more basic statistical tests we can run is to see if the expected value (mean) of a sample from a distribution is equal to a given value. For example:

```
>>> ss.ttest_1samp([1,2,3,4,5], 3)
Ttest_1sampResult(statistic=0.0, pvalue=1.0)
```

This tells us that we were completely accurate: 3 is exactly equal to the mean of the provided data. In other words, the *null hypothesis* assuming that no difference exists between the mean of the list and the value given cannot be rejected. We can also access the outcomes individually by indexing into the result (e.g., assigning the outcome to `res` and running `res[1]` gives us the p-value). If you feel that your knowledge of this test is rusty, try running the code yourself with different means but similarly simple lists to observe the effect it has on the outcome.

If we have two samples, then we can also run a t-test to see if they have the same expected value; again, the null hypothesis assumes that they will be the same. If the samples are known to be *related* (e.g., they are exam scores `marks1` and `marks2` for the same set of students), we can run:

```
>>> ss.ttest_rel(marks1, marks2)
```

If instead the samples are known to be independent, meaning the populations contributing to them are non-overlapping (e.g., they are exam scores `marks_m` and `marks_f` for male and female students), we can run:

```
>>> ss.ttest_ind(marks_m, marks_f)
```

This will tell us whether or not the expected value for these samples differs significantly across the two populations. If the returned `pvalue` is high, meaning over 0.01 or 0.05, we cannot reject the null hypothesis (which assumes they do not differ), but if it is low we can.

Task 2. Go back to the random samples you created in Task 1. What would you expect their respective means to be? Do you think they would be the same or different? Put your hypotheses in a comment and add code to `random_graph.py` to test them.

Solution 2. The mean for both should be 250, since they're randomly sampled. The code to test this is:

```
print ss.ttest_1samp(x, 250)
print ss.ttest_1samp(y, 250)
print ss.ttest_ind(x, y)
```

An independent t-test is used as the sampling for each was independent. These results will vary depending on the samples, but the p-value should be quite high for all of them and the statistic quite low (indicating that we are not far off from the true mean).

2.2 Correlation

It is often helpful to go beyond expected value, and instead see if two distributions are *correlated*, meaning values in one distribution can help you predict the value in another.

One way of indicating whether or not a correlation exists is Pearson's correlation coefficient, which we can run as follows:

```
>>> ss.pearsonr(dist1, dist2)
```

This returns two values: the correlation coefficient, and the p-value. As in previous tests, the p-value indicates the statistical significance of the answer. The correlation coefficient can range from -1 to $+1$, and indicates the level and type of correlation: the closer it is to either side the stronger the correlation. For example, if we run the test on identical distributions, we see:

```
>>> ss.pearsonr(range(1000), range(1000))
(1.0, 0.0)
```

The p-value indicates that the result is highly statistically significant, and the coefficient indicates a strong positive correlation; this should not be surprising, given that the distributions are identical. Similarly, if we run it on exactly inverse distributions, we see:

```
>>> ss.pearsonr(range(1000), range(1000,0,-1))
(-1.0, 0.0)
```

On the other hand, if we run it on completely random distributions, we see:

```
>>> ss.pearsonr(random.sample(range(1000),500),random.sample(range(1000),500))
(-0.018902209649921779, 0.67328339090860001)
```

The p-value you see here will vary as you run the test (since of course the samples are random), but it is very unlikely to see an indication of statistical significance. You are similarly likely to always see a correlation coefficient close to 0 (indicating no correlation); again, this should not be surprising given the nature of the distributions.

Finally, to get not only the correlation across the distributions but also how to actually obtain one from the other, we could run a *linear regression*. This gives us the slope a and intercept b of the line such that the values obtained by running $ax + b$ (where x ranges across the values in the first distribution) represent the best-fit line for the data in the second distribution. For example, if we again run the test on identical distributions, we see:

```
>>> ss.linregress(range(1000),range(1000))
LinregressResult(slope=1.0, intercept=0.0, rvalue=1.0, pvalue=0.0, stderr=0.0)
```

This tells us that the line is $y = x$, and that this line perfectly fits the data; again, this should not be surprising given that the distributions are identical. Similarly, if we again use the inverse distributions, we see:

```
>>> ss.linregress(range(1000), range(1000,0,-1))
LinregressResult(slope=-1.0, intercept=1000.0, rvalue=-1.0, pvalue=0.0, stderr=0.0)
```

This tells us that the line is $y = -x + 1000$, and that again this line perfectly fits the data.

Task 3. In `random_graph.py`, print the outcome of a linear regression for the random samples you created in Task 1, and collect in two variables the slope and intercept. In particular, describe in a comment the meaning of the regression as evidenced by the p-value and correlation coefficient. Is this what you expected?

Solution 3. The code for this is:

```
res = ss.linregress(x, y)
print res
slope = res[0]
intercept = res[1]
```

Again, the exact results will vary based on the sample. Typically there should be a quite large p-value, indicating that we cannot reject the null hypothesis that there is no linear relationship, and a very small correlation coefficient, indicating no correlation between the two distributions. This should be expected given that they are completely random samples.

3 Graphing with `matplotlib`

The core graphing package in Python is `matplotlib`. This is not an overly nice package to work with, so it is perhaps not surprising that many libraries exist online that form a *wrapper* around it, meaning they use it but provide a nicer interface to the user. Indeed you may remember that `pandas` has the ability to create plots, which it turns out is because it provides a wrapper around `matplotlib`. Another approach some programmers take (including myself) is to perform computations in Python but then switch to another language, like R, in order to create graphs. Thus, while we cover the basics of `matplotlib` here, we are certainly not endorsing it and you are welcome to use any other graphing solution with which you feel comfortable.

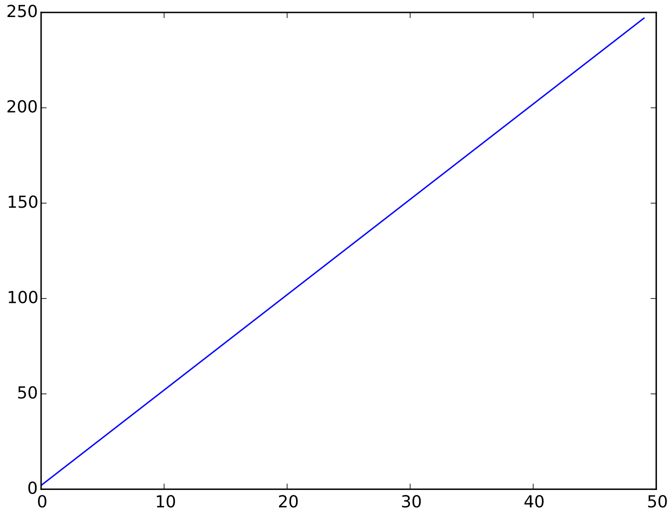
As with previous packages, there is a recommended abbreviation when importing `matplotlib`; this is `import matplotlib.pyplot as plt`.

3.1 Basic plots

Arguably the simplest graph we can create is a line graph. For example, suppose we want to graph the line $y = 5x + 2$. To do this, we can run:

```
xaxis = range(50)
yaxis = [5*x + 2 for x in xaxis]
plt.plot(xaxis, yaxis)
```

Now what? If we want to see the graph directly, we can run `plt.show()`; this will open a window with the graph in it. If we instead want to save it to a file, we can run `plt.savefig('line.pdf')` (where you can replace `line.pdf` with any filename you like). Running this code will add the graph into the current working directory. Either way, the graph looks about as exciting as we'd expect:



While the default line is solid blue, there are ways to change this; e.g., if we wanted a dashed red line we could run:

```
plt.plot(xaxis, yaxis, color='red', linestyle='dashed')
```

Again, our goal here is not to enumerate all such manipulations available in `matplotlib`, so we refer to the documentation (`https://matplotlib.org/`) for a complete list.

We can also have multiple lines on the same plot by simply running `plt.plot` again (or running it in a loop). We can also add nice features such as axis labels, a title, a legend, etc., and we can manually set the limits of both the x-axis and the y-axis. For example, to create a graph with four lines, we could run:

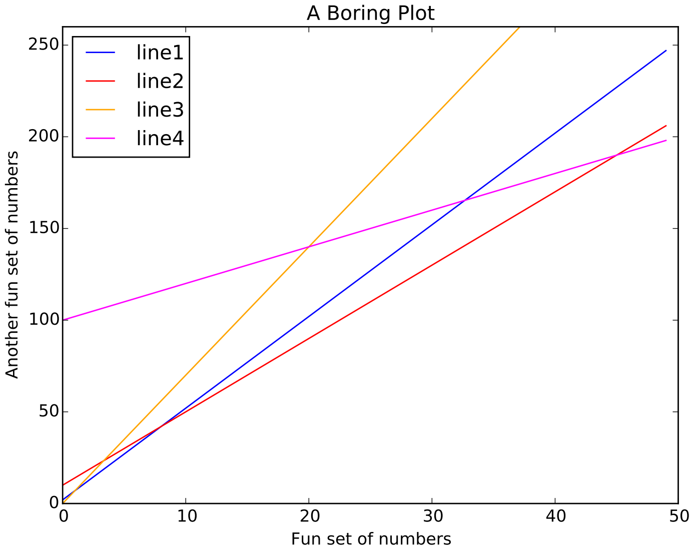
```
xaxis = range(50)
yaxes = []
yaxes.append([5*x + 2 for x in xaxis])
yaxes.append([4*x + 10 for x in xaxis])
yaxes.append([7*x for x in xaxis])
yaxes.append([2*x + 100 for x in xaxis])

labels = ['line1', 'line2', 'line3', 'line4']
colors = ['blue', 'red', 'orange', 'magenta']

for (i,yaxis) in enumerate(yaxes):
    plt.plot(xaxis,yaxis,color=colors[i],label=labels[i])

plt.ylim(0, 260)
plt.xlabel('Fun set of numbers')
plt.ylabel('Another fun set of numbers')
plt.legend(loc='best')
plt.title('A Boring Plot')
plt.savefig('lines.pdf')
```

This `for` loop plots each of the lines with its associated color and label; you could also imagine storing this related data in a dictionary. It also sets explicit limits for the y-axis (`plt.ylim`), labels for the x-axis (`plt.xlabel`) and y-axis (`plt.ylabel`), a title for the plot (`plt.title`), and puts the legend wherever `matplotlib` thinks is best (`plt.legend`). It gives us the following even more exciting graph:

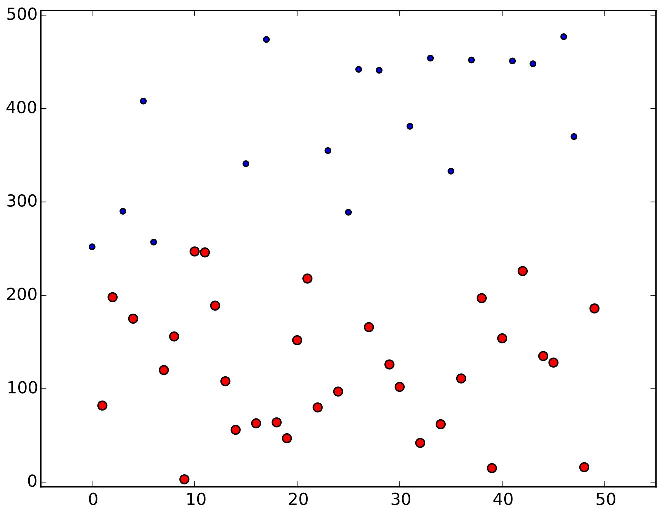


3.2 Scatter plots

Sometimes, especially for data that is not necessarily so related, we want to produce a *scatter* plot, in which the points are drawn but not connected. For example, consider the following code:

```
xaxis = range(50)
yaxis = random.sample(range(500),50)
colors = []
sizes = []
for y in yaxis:
    if y > 250:
        colors.append('blue')
        sizes.append(15)
    else:
        colors.append('red')
        sizes.append(40)
plt.scatter(xaxis,yaxis,s=sizes,c=colors)
plt.xlim([-5,55])
plt.ylim([-5,505])
plt.savefig('points.pdf')
```

This graphs the data points according to the colors (blue if the value is bigger and red if it is smaller) and sizes (smaller if the value is bigger and bigger if it is smaller) that we have specified in the appropriate lists; again, this is data you could also imagine storing in a dictionary. It produces something like the following graph:



Task 4. In `random_graph.py`, plot the two random samples you created in Task 1 against each other using a scatter plot. Add the best-fit line according to the regression you conducted in Task 3, and save the graph in a file `plot.pdf`.

Solution 4. The code for this (and all the previous tasks) is in `random_graph.py`. The graphing part is:

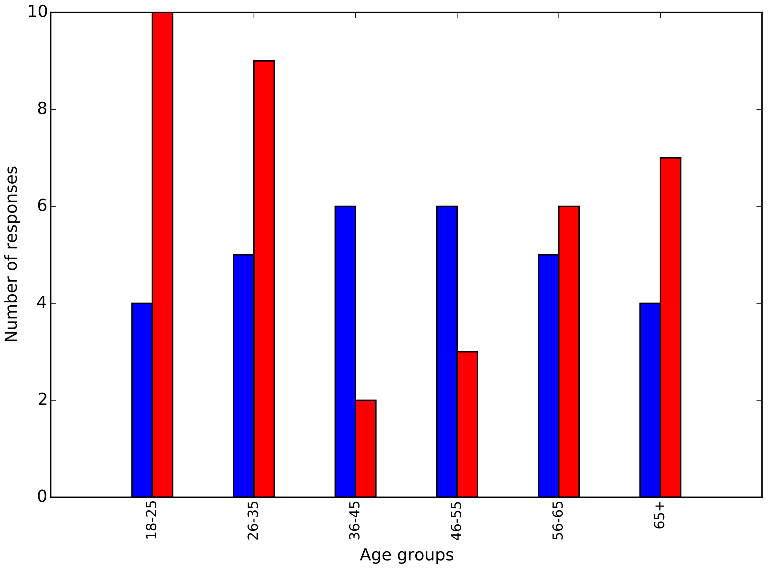
```
plt.scatter(x,y)
plt.plot(x, [slope*x_val + intercept for x_val in x])
plt.xlim([0,500])
plt.ylim([0,500])
plt.savefig('plot.pdf')
```

3.3 Bar graphs

Finally, sometimes we might want to sort data categorically, or put it into buckets; e.g., we could count the number of people within a given age group who answered a question a certain way. For this we could imagine using a bar graph. If within each category we have further subcategories, we could imagine using a *stacked* or a *clustered* bar graph. For example, consider the following code:

```
xaxis = ['18-25', '26-35', '36-45', '46-55', '56-65', '65+']
yaxis_male = [4, 5, 6, 6, 5, 4]
yaxis_female = [10, 9, 2, 3, 6, 7]
plt.bar(map(lambda x : x - 0.1, range(len(xaxis))), yaxis_male,
        align='center', color='blue', width=0.2)
plt.bar(map(lambda x : x + 0.1, range(len(xaxis))), yaxis_female,
        align='center', color='red', width=0.2)
plt.xlabel('Age groups')
plt.xticks(range(len(xaxis)), xaxis, fontsize=10, rotation=90)
plt.ylabel('Number of responses')
plt.tight_layout()
plt.savefig('figs/bars.pdf')
```

The use of `map` introduces an offset in the clustered bars, so that they don't overlap (this is one use case where `matplotlib` is particularly clunky). Otherwise, the code graphs the bars, puts the name of the categories on the x-axis with a rotated label (`plt.xticks`), and crops the overall figure to be close to the graph (`plt.tight_layout()`). It produces the following graph:

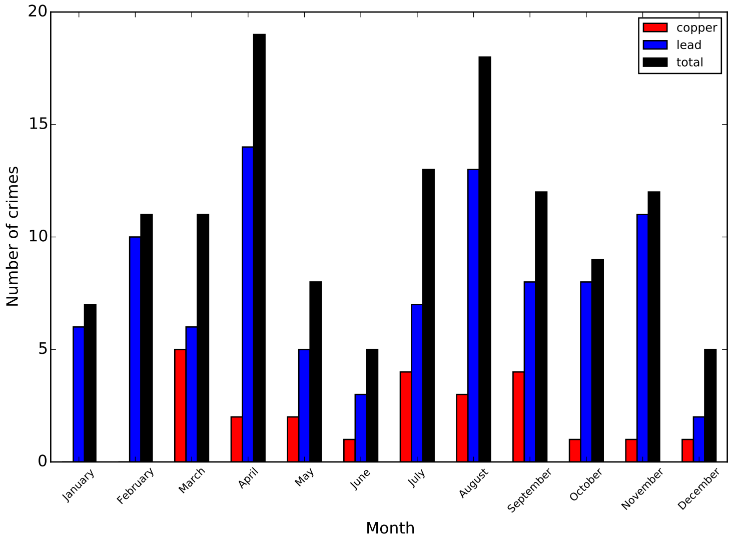


4 Putting It All Together

In Labs 04 and 05, we counted the number of metal thefts matching a certain pattern; i.e., occurring in a certain month/season or stealing a certain type of metal. The only way we had to report our findings back then was by printing them out in the command line, but now we thankfully have more meaningful and communicative tools available.

Task 5. First, copy into a single file `church_theft.py` either your or my solutions from Labs 04 and 05 for counting the number of thefts stealing lead and the number stealing copper (ignoring subtypes like flashing), and for counting the number of crimes per month (ignoring seasons). Combine them to create a clustered bar graph in a file `months.pdf` where the x-axis represents the months in a year (identified by their name, not their number), and the y-axis represents the number of crimes that occurred in that month. There should be a legend, both axes should be labeled, and there should be three bars: one for lead, one for copper, and one for the total number of crimes.

Solution 5. The code for this is in `church_theft.py`. The resulting graph is:



5 Handing In

As always, feel free to continue trying out the various concepts we've covered in this lab. Once you are done, add the files into a `lab08` directory; this should mean adding:

- `random_graph.py`
- `church_theft.py`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.