Generating random numbers

• Allows us to create baseline values

• We can shuffle datasets, split them into random groups

• Compare against random samples
Commonly used functions

```python
import random

# random number between 0 and 1
print random.random()

# random integer between the limits
# including the two limits!
print random.randint(1, 100)

# random number from a normal (gaussian)
# distribution of mean=1,
# and standard deviation=10)
print random.gauss(mu=1, sigma=10)

python -tu C:\cygwin\home\ialbert\sources\ialbert-web\ppt\week7\lecture13\randnums.py 2>&1 \ returned 0

0.68521959858
21
9.13325179721
```
Generating random numbers

```python
import random

# seed allows you to generate the
# same sequence of random numbers repeatedly
random.seed(39929)

def generate(x):
    return random.random()

data = range(10)
values = map(generate, data)

print values[:3]
```

```
[0.59585819492243042, 0.93760022126109865, 0.036526830134548383]
```
Pigeonhole principle

Occurs when distributing a larger set over a smaller one

10 pigeons over 9 boxes $\rightarrow$ some box will end up with more than one pigeon

Other “pigeonhole” problems are a lot less obvious
1. Take a list of numbers (size N)

2. Go over each element and swap it with any other element
Shuffling three elements

Some end configurations are more frequent than others
Pigeonhole problem

- Total possible end configurations: $N!$
- Total possible paths: $N^N$

$N^N > N!$

The incorrect algorithm described used to be *widely circulated*.

**Paper:** *How We Learned to Cheat at Online Poker: A Study in Software Security*
The correct shuffling

- Fisher-Yates $\rightarrow$ 1938 complexity of $O(N^2)$
- Durstenfeld-Knuth $\rightarrow$ 1964 complexity of $O(N)$
Unexpected consequence

Python’s random number generator
Mersenne Twister by Mashumoto-Nishimura (1997)
Period of $2^{19937}$

List permutations $\rightarrow N!$

If $N! > 2^{19937}$ it becomes impossible to find all permutations of a list

(we can still find a lot of them, up to the period of the generator)
Computing the largest size

```python
import math

period = 2 ** 19937

print(math.factorial(10))

# print period
```

```
3628800
```
Sampling – pick one element at random

```python
1 import random
2 data = xrange(10)
3 print random.choice(data)
```
Sampling – without replacement
(unique elements)

```python
import random

data = xrange(10)

for i in range(3):
    samp = random.sample(data, 3)
    print samp
```

```
[4, 8, 1]
[6, 4, 3]
[1, 5, 0]
```
Sampling – with replacement

```python
import random

data = ['A', 'B', 'C', 'D']

def random_wr(x):
    return random.choice(data)

samp = map(random_wr, range(3))

print samp
```

Output:
```
['B', 'D', 'B']
```
Homework 23

Generate the histogram shown on the next page. The data \( \rightarrow \) random gaussian distribution. Mean=100 Standard deviation \( \sigma=10 \).

Vertical lines are at mean-\( \sigma \), mean and at mean+\( \sigma \) positions.

How many datapoints did you need to get a smooth distribution (approximation)?

What percent of your generated values are less than one standard deviation (\( \sigma \)) away from the mean \( (\text{abs(mean-x)}<\sigma) \)?
Homework: desired output
Week 12 - Lecture 24

István Albert

BMB, Bioinformatics
Normal distribution

one standard deviation

68.3%

95.4%

99.7%
Why do so many measures take the shape of a normal distribution?

Central Limit Theorem

The sum of a large number of independent random variables is distributed approximately normally.
When are two values different in a reliable (statistically significant) way?

**Note**: there is a fundamental difference between biologically significant and statistically significant.

P-value → there is no difference.
Statistical significance does not mean accuracy or correctness

General problems when using p values

1. Reporting very large numbers $p = 1E-30$
   the chance of being wrong is claimed to be: $1/1000000000000000000000000000000$

2. Smaller significance values presented as more convincing results
Accuracy and Precision

This is a random-like pattern, neither precise nor accurate. The darts are not clustered together and are not near the bull's eye.

This is a precise pattern, but not accurate. The darts are clustered together but did not hit the intended mark.

This is an accurate pattern, but not precise. The darts are not clustered, but their 'average' position is the center of the bull's eye.

This pattern is both precise and accurate. The darts are tightly clustered and their average position is the center of the bull's eye.
Numpy and Scipy

- **Numpy** – fast numerical computation library

- **Scipy** (relies on numpy) – contains more specialized modules, for example statistical functions that make use of numpy
For details on statistical functions use the scipy reference

• Search for the scipy documentation

• Large number of statistical functions and methods

• `scipy.stats` module \(\rightarrow\) contains statistical functions
SciPy is open-source software for mathematics, science, and engineering. It is also the name of a very popular conference on scientific programming with Python. The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The SciPy library is built to work with NumPy arrays, and provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization. Together, they run on all popular operating systems, are quick to install, and are free of charge. NumPy and SciPy are easy to use, but powerful enough to be depended upon by some of the world’s leading scientists and engineers. If you need to manipulate numbers on a computer and display or publish the results, give SciPy a try!

SciPy is a community effort. We seek volunteers at all levels of ability to work on the project, from coding and packaging to documentation, tutorials, recipes, and the web site. Visit the Developer Zone if you are interested in helping out (or if you have bug reports).

Python and Scientific Computing

NumPy and SciPy are two of many open-source packages for scientific computing that use the Python programming language. This website, together with other subdomains of the scipy.org domain, serves as a portal for all scientific computing with Python, not just NumPy and SciPy. The index under Topical Software in the navigation bar lists these domains and other destinations for scientific software using Python.

Good places to start to learn more about SciPy:
### Scipy Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`cov(m, y, rowvar, bias)</td>
<td>Estimate the covariance matrix.</td>
</tr>
<tr>
<td><code>corcoef(x, y, rowvar, bias)</code></td>
<td>The correlation coefficients formed from 2-d array x, where the rows are the observations, and the columns are variables.</td>
</tr>
<tr>
<td><code>f_oneway(*args)</code></td>
<td>Performs a 1-way ANOVA.</td>
</tr>
<tr>
<td><code>pearsonr(x, y)</code></td>
<td>Calculates a Pearson correlation coefficient and the p-value for testing</td>
</tr>
<tr>
<td><code>spearmanr(x, y)</code></td>
<td>Calculates a Spearman rank-order correlation coefficient and the p-value</td>
</tr>
<tr>
<td><code>pointbiserialr(x, y)</code></td>
<td>Calculates a point biserial correlation coefficient and the associated p-value.</td>
</tr>
<tr>
<td><code>kendalltau(x, y)</code></td>
<td>Calculates Kendall's tau, a correlation measure for ordinal data</td>
</tr>
<tr>
<td><code>linregress(*args)</code></td>
<td>Calculate a regression line</td>
</tr>
<tr>
<td><code>tttest_1samp(a, popmean[, axis])</code></td>
<td>Calculates the T-test for the mean of ONE group of scores a.</td>
</tr>
<tr>
<td><code>tttest_ind(a, b[, axis])</code></td>
<td>Calculates the T-test for the means of TWO INDEPENDENT samples of scores.</td>
</tr>
<tr>
<td><code>tttest_rel(a, b[, axis])</code></td>
<td>Calculates the T-test on TWO RELATED samples of scores, a and b.</td>
</tr>
<tr>
<td><code>ksctest(rv, cdf, </code>kwds<code>, [args, N, ...])</code></td>
<td>Perform the Kolmogorov-Smirnov test for goodness of fit</td>
</tr>
<tr>
<td><code>chisquare(f_obs[, f_exp])</code></td>
<td>Calculates a one-way chi square test.</td>
</tr>
<tr>
<td><code>ks_2samp(data1, data2)</code></td>
<td>Computes the Kolmogorov-Smirnov statistic on 2 samples.</td>
</tr>
<tr>
<td><code>mannwhitneyu(x, y, use_continuity)</code></td>
<td>Computes the Mann-Whitney rank test on samples x and y.</td>
</tr>
<tr>
<td><code>tiecorrect(rankvals)</code></td>
<td>Tie-corrector for ties in Mann Whitney U and Kruskal Wallis H tests.</td>
</tr>
<tr>
<td><code>ranksums(x, y)</code></td>
<td>Compute the Wilcoxon rank-sum statistic for two samples.</td>
</tr>
<tr>
<td><code>wilcoxon(x[, y])</code></td>
<td>Calculate the Wilcoxon signed-rank test</td>
</tr>
<tr>
<td><code>kruskal(*args)</code></td>
<td>Compute the Kruskal-Wallis H-test for independent samples</td>
</tr>
<tr>
<td><code>friedmanchisquare(*args)</code></td>
<td>Computes the Friedman test for repeated measurements</td>
</tr>
<tr>
<td><code>ansari(x, y)</code></td>
<td>Perform the Ansari-Bradley test for equal scale parameters</td>
</tr>
<tr>
<td><code>bartlett(*args)</code></td>
<td>Perform Bartlett's test for equal variances</td>
</tr>
<tr>
<td><code>levenc(*args, </code>kwds<code>)</code></td>
<td>Perform Levene test for equal variances</td>
</tr>
<tr>
<td><code>shapiro(x[, a, reta])</code></td>
<td>Perform the Shapiro-Wilk test for normality</td>
</tr>
<tr>
<td><code>anderson(x[, dist])</code></td>
<td>Anderson-Darling test for data coming from a particular distribution</td>
</tr>
<tr>
<td><code>binom_test(x[, n, p])</code></td>
<td>Perform a test that the probability of success is p.</td>
</tr>
<tr>
<td><code>fligner(*args, </code>kwds<code>)</code></td>
<td>Perform Fligner’s test for equal variances</td>
</tr>
<tr>
<td><code>mood(x, y)</code></td>
<td>Perform Mood’s test for equal scale parameters</td>
</tr>
<tr>
<td><code>oneway(*args, </code>kwds<code>)</code></td>
<td>Test for equal means in two or more samples from the normal distribution.</td>
</tr>
<tr>
<td><code>glm(data, para)</code></td>
<td>Calculates a linear model fit ...</td>
</tr>
</tbody>
</table>
Mean and median

```python
import numpy

x = [1, 1, 1, 2, 3, 4, 5, 6, 1000]

print 'Mean    = %4.2f' % numpy.mean(x)
print 'Median  = %4.2f' % numpy.median(x)
print 'Stdev   = %4.2f' % numpy.std(x)
```

```
Mean    = 113.67
Median  = 3.00
Stdev   = 313.37
```
Trimmed mean and median

```python
import numpy
from scipy import stats

x = [1, 1, 1, 2, 3, 4, 5, 6, 1000]
lims = (0, 7)

print 'Mean  = %4.2f' % numpy.mean(x)
print 'Median = %4.2f' % numpy.median(x)
print 'TMean  = %4.2f' % stats.tmean(x, lims)
print 'Stdev  = %4.2f' % numpy.std(x)
print 'TStdev = %4.2f' % stats.tstd(x, lims)
```

Command Output:

```
Mean  = 113.67
Median = 3.00
TMean  = 2.88
Stdev  = 313.37
TStdev = 1.96
```
Null hypothesis – no change

• Expect no change between the samples

• $p \rightarrow$ p-value, the likelihood that the null hypothesis is true

• $1-p$ is the chance that the null hypothesis is not true, there is a difference between the measures
import numpy, random
from scipy import stats

def create(mean, sigma, size=100):
    def rnd(x):
        return random.gauss(mu=mean, sigma=sigma)
    return map(rnd, xrange(size))

data = create(10, 2, 100)

print numpy.mean(data)
print numpy.std(data)
Interval estimates

The mean or standard deviation are **point estimates** → one value

Confidence intervals inform of the limits the values may take within a certain confidence interval → 95% of the data will be between

[low limit, high limit]
```python
def create(mean, sigma, size=100):
    def rnd(x):
        return random.gauss(mu=mean, sigma=sigma)
    return map(rnd, xrange(size))

data = create(2, 3, size=10)

print numpy.mean(data)

# a tuple triplet that contains the
# confidence intervals for mean, var and standard deviations
print stats.bayes_mvs(data, alpha=0.95)[0]
```

```
2.07004529514
(2.0700452951380242, (0.53302233685248068, 3.6070682534235674))
```
t-tests that we will cover

1. Comparing **sample** and **population** means

2. Comparing two **independent samples**

3. Comparing two **related samples**
t-test for means

We know the population mean value.

We perform $N$ measurements (sample).

Q: What is the probability that our sample has the same mean value as the population mean value?

```python
popmean = 2.0
genmean = popmean - 0.1
data = create(genmean, 3, size=100)

score, pval = stats.ttest_1samp(data, popmean)

# returns the score and p-value
print 'Pval=%0.2g, Significant=%s' % (pval, pval < 0.01)
```

```
Pval=0.76, Significant=False
```
Generated data $\rightarrow$ replication

```
6    return random.gauss(mu=mean, sigma=sigma)
7    return map(rnd, xrange(size))

9        popmean = 2.0
10       genmean = popmean - 0.1

12    for i in range(3):
13       data = create(genmean, 3, size=10)
14       score, pval = stats.ttest_1samp(data, popmean)
15       print 'Pval=%4.2g, Significant=%s' % (pval, pval < 0.1)
```

```
Pval=0.77, Significant=False
Pval=0.27, Significant=False
Pval=0.072, Significant=True
```
Assume that the population mean is 2.0. Generate samples with a mean of popmean - 0.1. (set the random seed to a constant)

• What is the smallest sample size (approximate) that generates 95% of the mean values between at least 1.5 and no more than 2.5 if the standard deviation (sigma) is set to 2 then 3

• What is the largest sigma value that allows you to reliably detect a difference at 1% significance level between the population mean and the sample mean if the sample size is set to 100.

• Turn off the random seed line. Do your results change?