CSCI2202: Lecture 9 Randomness and Probability

Finlay Maguire (finlay.maguire@dal.ca)
TA: Ehsan Baratnezhad (ethan.b@dal.ca)
TA: Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- Numpy basics and gotcha
- Randomness
- Pseudorandom Number Generators
- Numpy PRNG functions
  - Using PRNGs to estimate values (Monte Carlo)
- Probability Distributions
  - Estimating outcomes by sampling from distributions
- Optimisation
  - Grid-search vs Random-search
- Simulation
  - Calculating p-values by simulating Null distribution
  - Bootstrapping to estimate confidence intervals

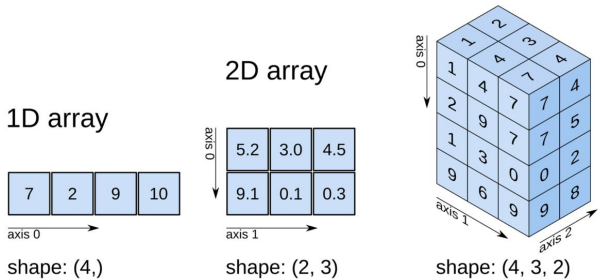
Numpy

NumPy (**Num**erical **Py**thon): numerical/array library that forms basis of most of scientific python

Core `ndarray` object that supports arrays with arbitrary numbers of dimensions/axes.

Operates similar to nested lists (index, slicing, nested dimensions) but with many special numerical methods.

Limitations for speed: elements only 1 type, semi-mutable (change values/shape but not number of entries), rectangular (i.e., each row must have same number of columns)

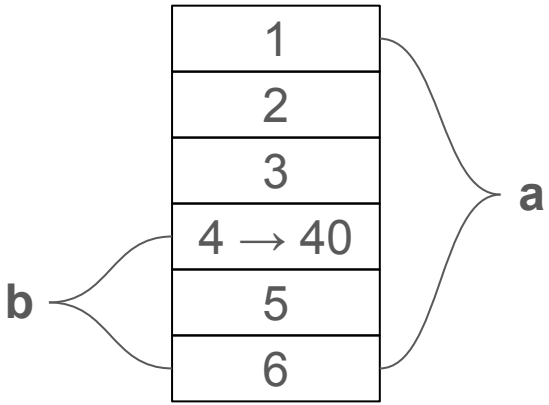


https://numpy.org/doc/stable/user/absolute_beginners.html

```python
import numpy as np

np.zeros(3) # array([0., 0., 0.])

np.ones(2) # array([1.,1.])

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(a)

array([[1, 2, 3],

       [4, 5, 6],

       [7, 8, 9]])

a[0,1] # 2

a[:, 2] # array([2, 5, 8])

a.shape # (3, 3)

a.reshape(1, 9) # array([[1, 2, 3, 4, 5, 6, 7, 8,
9]])

a.sum() # np.int64(45)
```

Numpy/Pandas: Views vs Copies

- Space efficiency: NumPy tries to avoid making copies of data
- Many NumPy operations e.g., subset of an array create a **VIEW**
- Like alias but to just a portion of the original array
- Changing a view will change part of the original array
- If want a **COPY** then make an explicit copy (slow): `ndarray.copy`
- As Pandas is built on numpy it often does the same thing (one of the main sources of “warnings” in Pandas).




```
a = np.array([1, 2, 3, 4, 5, 6])

a

array([1, 2, 3, 4, 5, 6])

b = a[3:]

b

array([4, 5, 6])

b[0] = 40

a

array([ 10,  2,  3, 40,  5,  6])
```

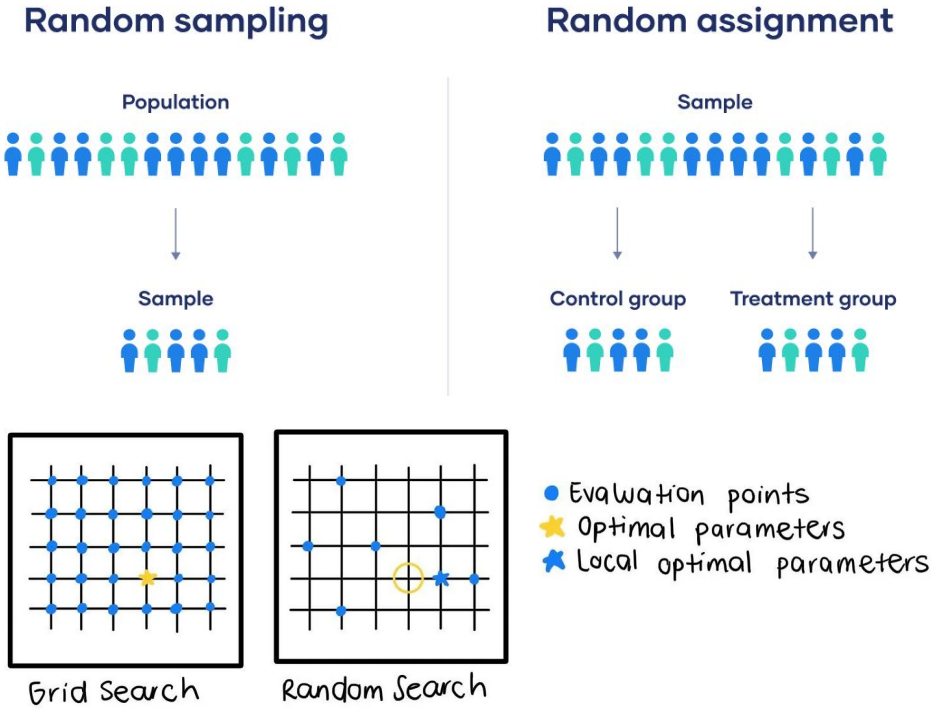
Randomness

Randomness is an essential tool in science

Almost every area of science makes use of randomness:

Sampling: Random subset of group can be used to create a smaller unbiased representation of group

Optimisation: Efficiently sampling possible values to find “best” combination of parameters (e.g., model fitting)



https://www.scribbr.com/methodology/random-assignment/,
https://engineering.rappi.com/bayesian-optimization-how-to-calibrate-the-hyper-parameters-of-computing-expensive-models-c3ca54b39dbb,

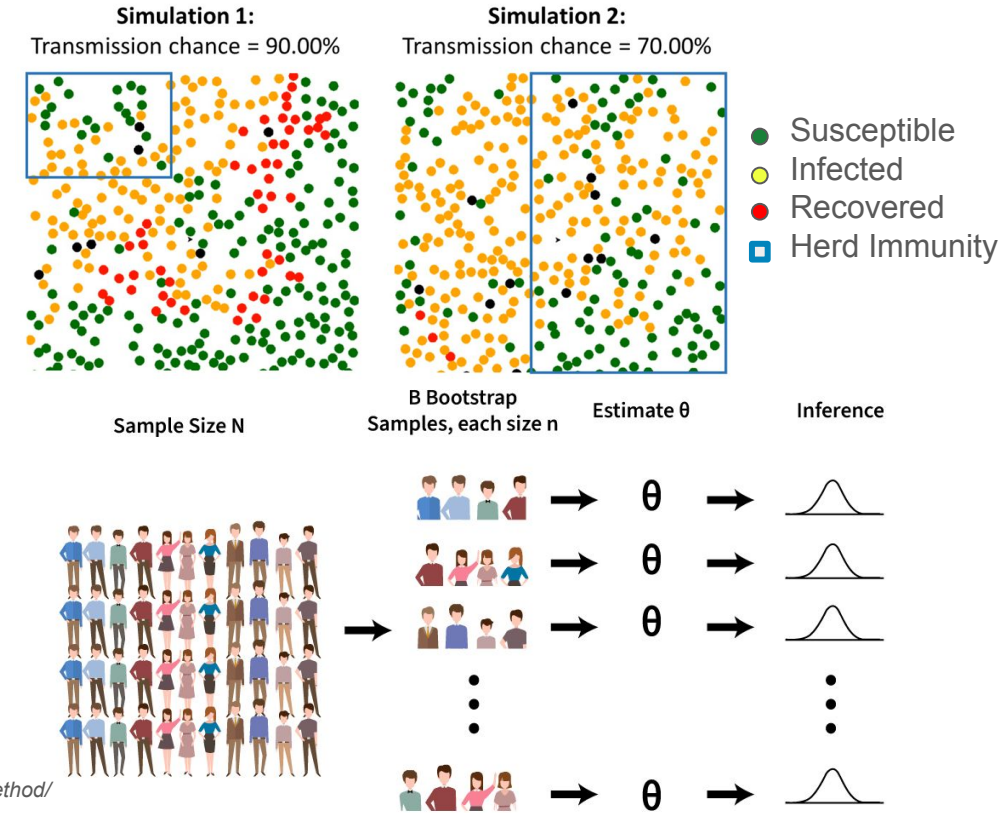
Randomness is an essential tool in science

Almost every area of science makes use of randomness:

Simulation: Model what different scenarios could look like to understand a system.

Estimation: Calculate something you can't directly measure through random sampling and simulation

(Many of these are actually the same thing!)



https://f1000research.com/articles/9-1198, https://www.geeksforgeeks.org/bootstrap-method/

(Pseudo)random Number Generation

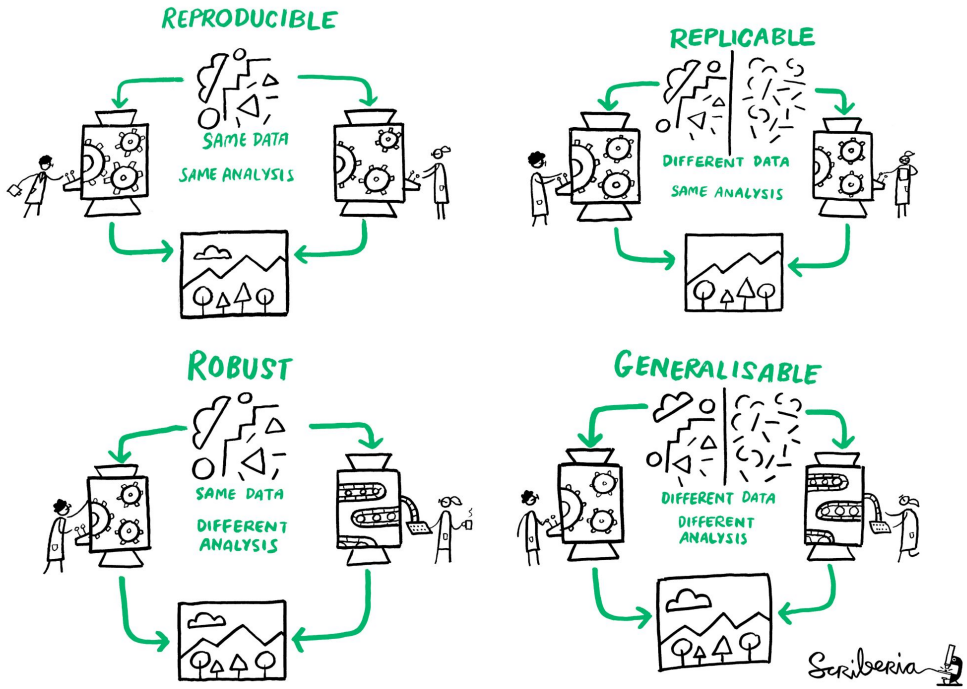
- True random numbers only from random information (e.g., hardware random physical processes - `secrets` module)
- Usually “pseudorandom” numbers using pseudorandom number generators (PRNGs).
- PRNGs **DETERMINISTICALLY** generate a series of random-appearing numbers based on a `seed` parameter
- Statistical tests to work out if random enough
- Linear Congruential Generator (LCG) one of simpler methods but has issues
- Numpy uses a Permutation Congruential Generator which does an additional permutation step

```
def lcg_prng(seed, size):
	state = seed
	multiplier = 6364136223846793005 # big prime numbers
	increment = 1442695040888963407 # big prime numbers
	modulo = 2 ** 64 # normalise to integer
	for i in range(size): # incrementally update seed/state
		state = (multiplier * state + increment) % modulo
		yield state

list(lcg_prng(42,5)) #[10481999410520546993,4159066171780167020]

list(lcg_prng(42,2)) #[10481999410520546993,4159066171780167020]

list(lcg_prng(10,2)) #[9743825058228238609,3298011952073619532]

list(lcg_prng(10,4)) # [9743825058228238609,3298011952073619532,
6582542797419383 47,4015636148853147230]
```

Reproducible research = setting a random seed

- For reproducibility: explicitly set this random seed (or python will choose a true random number as seed).

- `random`:
  - Built-in but relatively slow
  - Seed set: `random.seed`
- `np.random`
  - fast, optimised, numerical library
  - `np.random.default_rng(seed)`
  - `np.random.seed(seed) # deprecated`



Numpy random PRNG functions

Create a PRGN generator with a specific seed (ideal: big positive integer, practice: 42…).

Could implement `choice` with `integers`

Random can be scaled to arbitrary ranges:

`(high-low) * random(0 to 1) + low`

```
rng = np.random.default_rng(42)

rng.integers(low=0, high=2, size=3)

array([6, 2, 7])

rng.choice(['A', 'B', 'C'])

‘B’
```

```
rng.uniform(low=3.0, high=7.0, size=2)

array([5.57546048, 6.29104645])


rng.random(size=2) # ==uniform: 0-1.0

array([0.37079802, 0.92676499])


(7-3) * rng.random(size=2) + 3 # scaling

array([5.21833915, 3.25526902]) #3->7


a = np.arange(4) # array([0,1,2,3])
rng.shuffle(a)
a
array([2, 3, 0, 1])
```

Estimating hard to measure values with random numbers

Monte Carlo Estimation/Simulation

- Odds of winning roulette always playing red?
- Analytical probability calculation:
  - p(win|red) = red_slots / all_slots
  - p(win|red) = 18 / (18 * 2 + 2)
  - p(win|red) = 0.473
- What if the roulette wheel was hidden?
- What if you didn’t know the rules of how to calculate probability directly?
- Alternative: just play red a lot and see how often you win or lose.
- More times you play the better an estimate of the true p(win|red)
- Developed formally by Stanislaw Ulam for the Manhattan Project

Estimating π with random samples

area of circle / total area = π/4

area ~= number of random samples inside

π = (points inside / total points) * 4

More samples = more accurate estimate

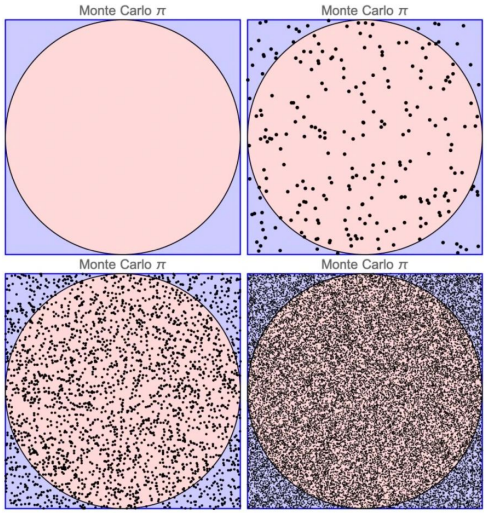


Figure 2: Squares with zero, 250, 2500 and 25000 points.

```python
def estimate_pi(num_points, rng):
    # Generate random points in a 2x2 square centered at origin
    points = rng.uniform(-1, 1, (num_points, 2))
    # Calculate distance from origin for each point
    distances = np.sum(points**2, axis=1)
    # Count points inside the unit circle (distance < 1)
    inside_circle = np.sum(distances <= 1)
    # Pi estimate: (points in circle / total points) * 4
    pi_estimate = (inside_circle / num_points) * 4
    return pi_estimate
estimate_pi(1, rng) # 4.0
estimate_pi(250, rng) # 3.152
estimate_pi(1_000_000, rng) # 3.142028
```

Sampling from probability distributions

Probability Distributions

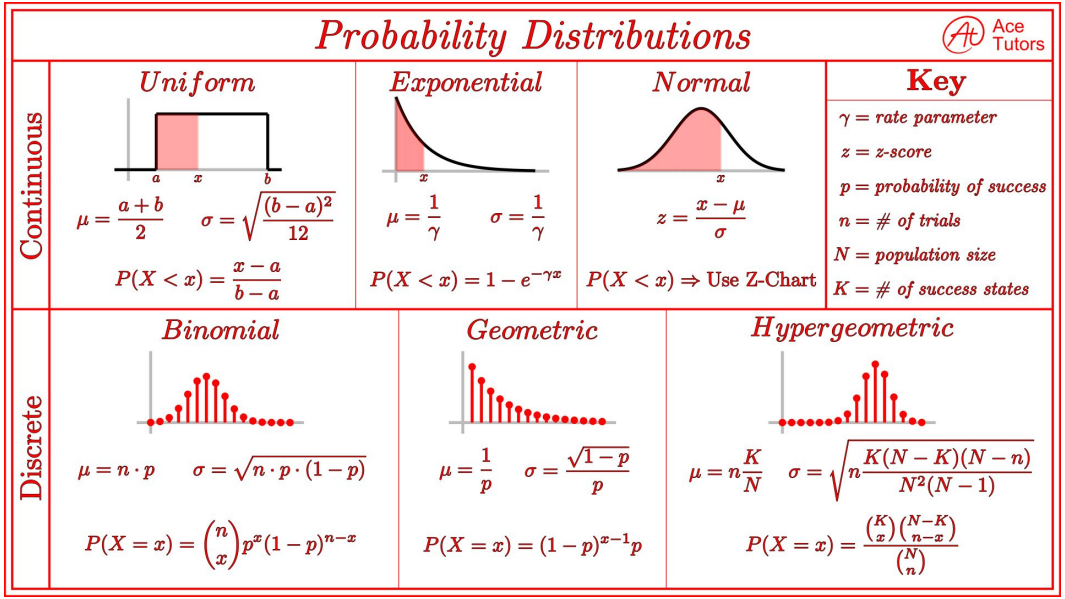
Random Variable: A variable whose value is determined by a random process. It can be:

- **Discrete**: Can only take specific values (like the number of heads in coin tosses) -
- **Continuous**: Can take any value within a range (like height or temperature)

Probability Mass Function (PMF): For discrete random variables, assigns probabilities to each possible value.

Probability Density Function (PDF): For continuous random variables, describes the relative likelihood of different values.

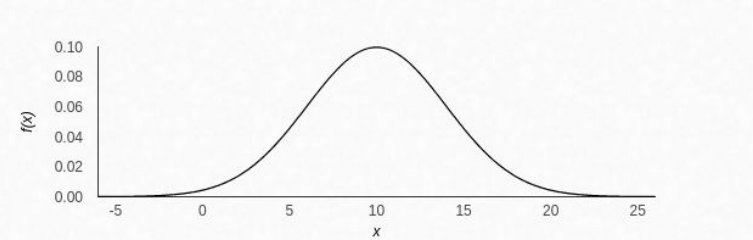
Cumulative Distribution Function (CDF): Gives the probability that a random variable is less than or equal to a certain value.



Sampling from specific distributions

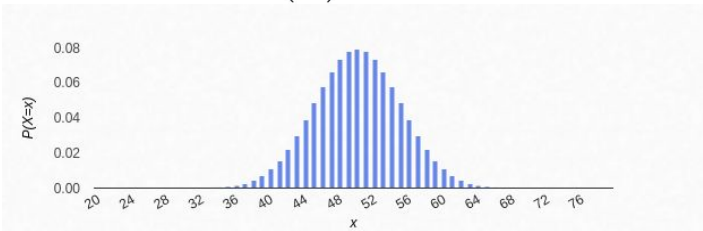
• **Normal/Gaussian**: sum of random variables tends to normal (central limit theorem), stay normal after many operations

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



• **Binomial**: binary successes of independent trials

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$



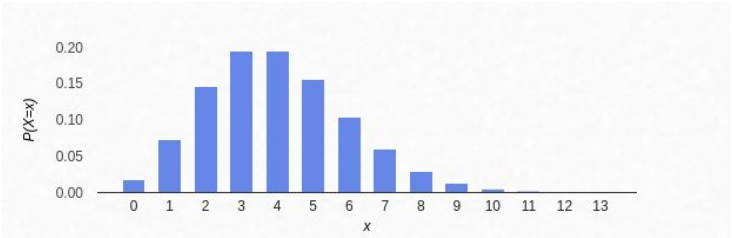

```
rng = np.random.default_rng(42)

rng.normal(loc=10, scale=4, size=3)

array([11.21886832,  5.84006358, 13.00180478]
```

```
rng.binomial(n=100, p=0.5) # 51

rng.binomial(n=100, p=0.5) # 56

rng.binomial(n=100, p=0.5) # 48
```

Sampling from specific distributions

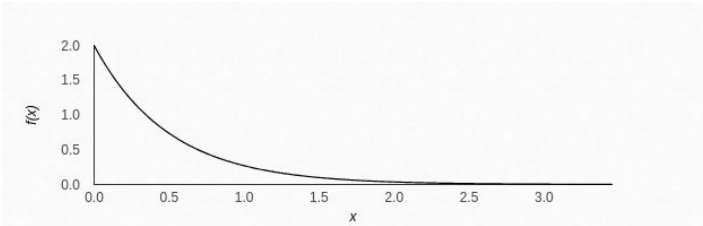
- **Poisson**: events occurring in a fixed interval

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$



- **Exponential**: time between events

$$f(x;\frac{1}{\beta}) = \frac{1}{\beta}\exp(-\frac{x}{\beta}),$$




```python
rng = np.random.default_rng(42)

rng.poisson(lam=4, size=3) #array([7, 3, 0])

rng.poisson(lam=4, size=3) # array([6, 4, 4])


rng.exponential(scale=2, size=4)

array([1.91320557, 1.24276617, 3.01592275, 4.05600316])


rng.exponential(scale=2, size=4)

array([0.66413763, 0.09950144, 1.84964447, 5.1068388 ])
```

Many distributions!

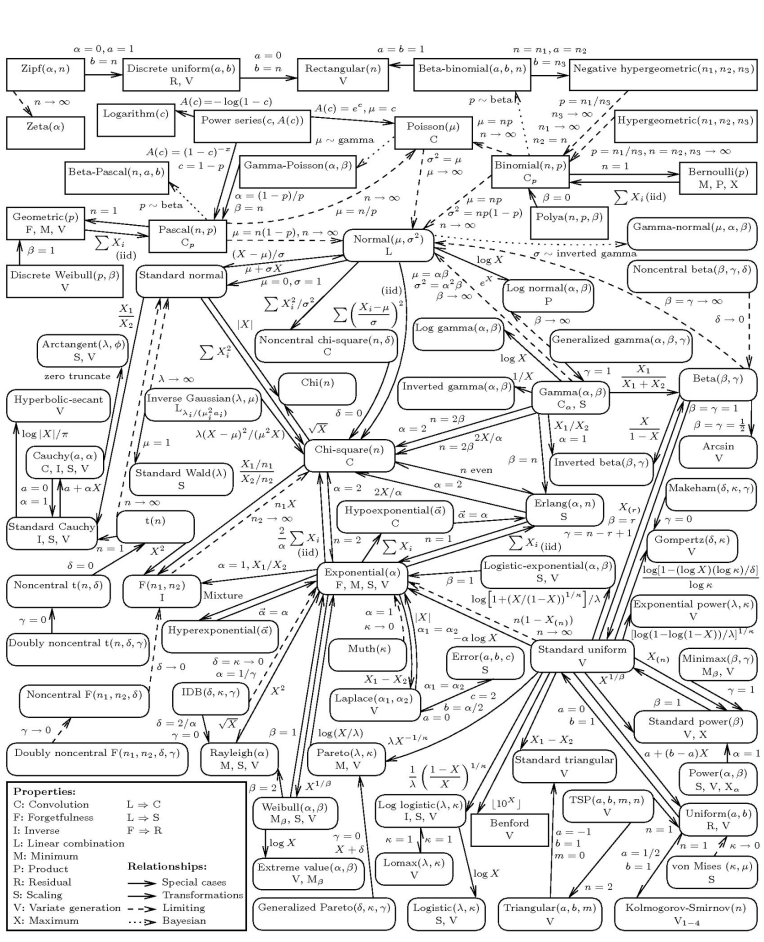
`np.random` has ~36 built-in distributions

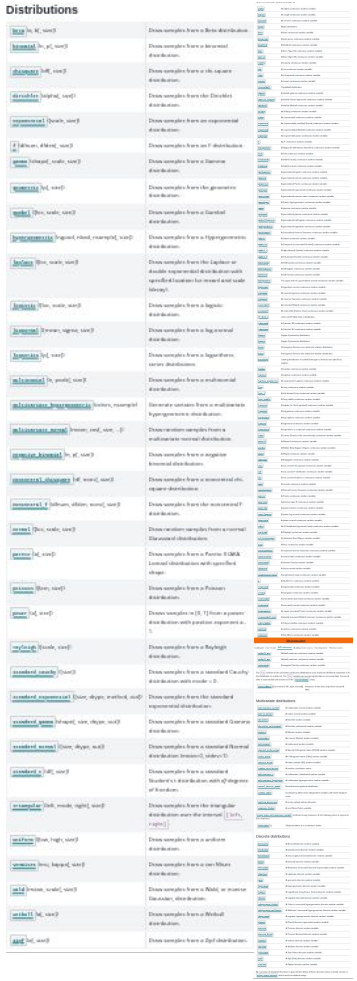
`scipy.stats` has > 120

Fun less-common distributions:

Cauchy - ratio of 2 normal random variables and has no mean or variance

Gumbel - used to calculate the probability of maximum values of random draws from other distributions





Understanding outcomes by sampling from a distribution

Let’s say we developed a vaccine

Clinical Trial showed that the vaccine prevented 70% of people exposed from getting infected.

If we gave this vaccine to 1000 people in the community and they were each exposed to the disease:

- How many of them would we expect to be protected?
- How variable would this be?

Can calculate this by randomly sampling from a binomial distribution!

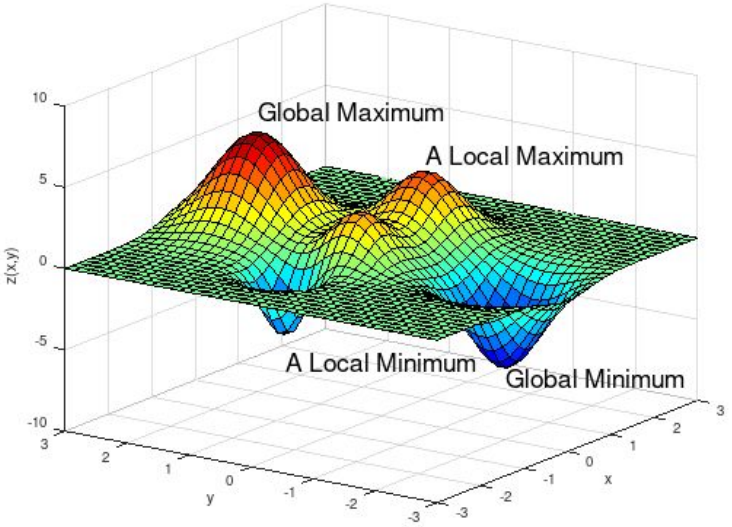
```
vaccine_protection = 0.7  # 70%

num_people = 1000  # Population size

num_simulations = 50  # Number of times to run the experiment

protected_counts = rng.binomial(n=num_people,

                                p=vaccine_effectiveness,

                                size=num_simulations)

median_protected = np.median(protected_counts)

std_protected = np.std(protected_counts)

print(f”{median_protected} +/- {std_protected}”)

693.5 +/- 16.569
```

Optimisation

Optimising functions is common in science & engineering

Scientific problems often involve finding the most efficient path, optimal configuration, or best parameters:

- Reducing drag in the design of a new boat propellor
- Finding the best evolutionary tree to explain DNA data
- Working out how to allocate vaccines to maximum effect

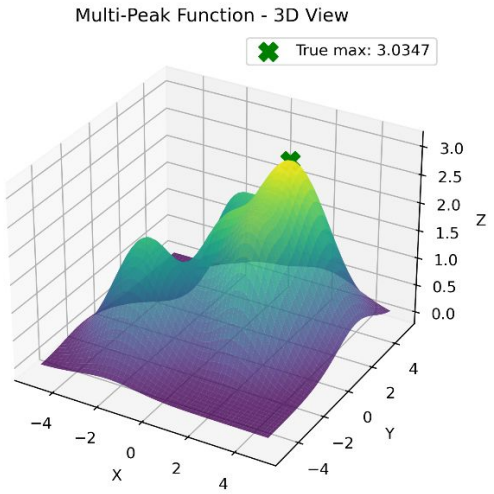


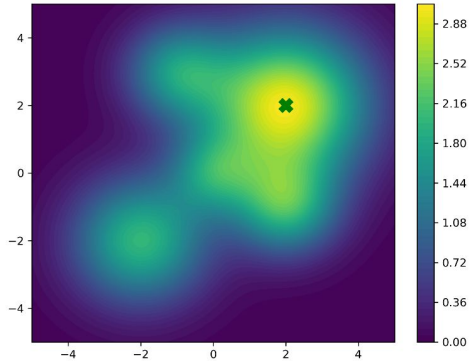
Optimisation without randomness means you can get stuck in local maxima/minima!

Functional optimisation

Problem: finding x and y with that maximise the value of a function (often unobservable directly)

```
def multi_peak_function(x, y):

    peak1 = 3 * np.exp(-0.2 * ((x - 2)**2 + (y - 2)**2))

    peak2 = 2 * np.exp(-0.3 * ((x + 2)**2 + (y + 2)**2))

    peak3 = 1.8 * np.exp(-0.5 * ((x - 2)**2 + (y + 1)**2))

    peak4 = 1.5 * np.exp(-0.5 * ((x + 1)**2 + (y - 3)**2))

    peak5 = 1.2 * np.exp(-0.7 * ((x - 0)**2 + (y - 0)**2))

    # Combine all peaks

    return peak1 + peak2 + peak3 + peak4 + peak5
```





Grid-Search

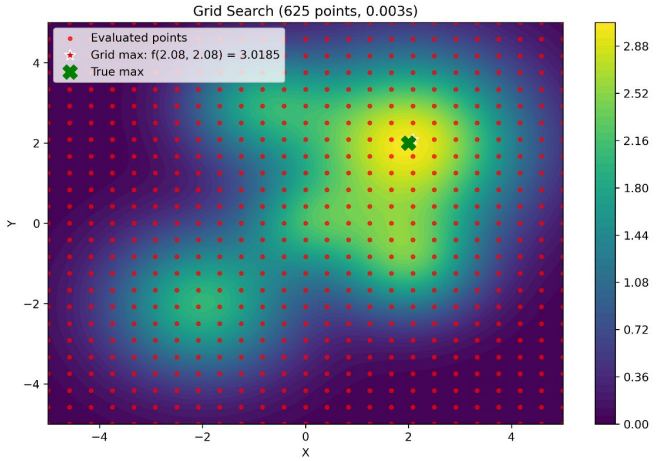
Problem: finding x and y with that maximise the value of a function (often unobservable directly)

Solution 1: try every possible x and y

- Infinite number of possibilities

Solution 2: systematically search x and y

- Many combinations but should be close



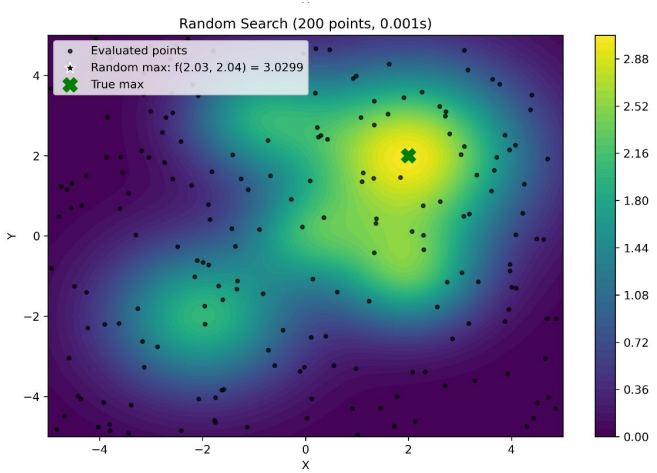

```
num_points = 25

x_grid = np.linspace(-5, 5, num_points)

y_grid = np.linspace(-5, 5, num_points)

max_value = float('-inf')

max_coords = (0, 0)

all_points = []

# Evaluate function at each grid point

for x in x_grid:

      for y in y_grid:

            z = multi_peak_function(x, y)

            all_points.append((x, y, z))

            if z > max_value:

                max_value, max_coords = z, (x, y)

all_points.sort(key=lambda point: point[2], reverse=True)

print(all_points[0])

(2.0833, 2.0833) = 3.0185
```

Random Search

Problem: finding x and y with that maximise the value of a function (often unobservable directly)

Solution 3: randomly try a number of possible values

- Much more efficient exploration of possible parameter space
- Not guaranteed to find optimal value

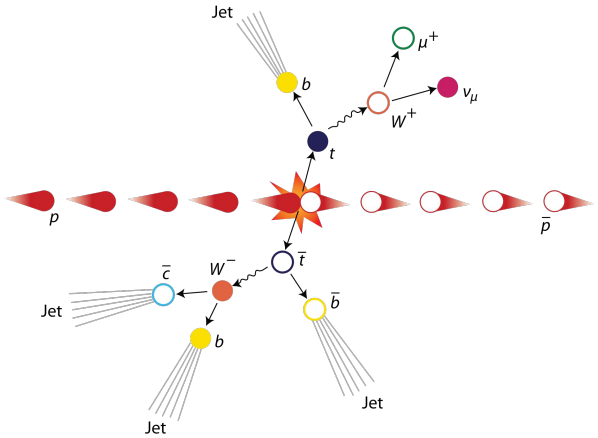


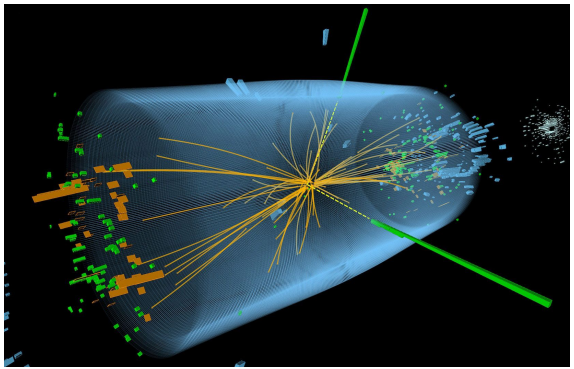

```
num_points = 200

x_random = np.random.uniform(-5, 5, num_points)

y_random = np.random.uniform(-5, 5, num_points)

Max_value, max_coords, all_points = float('-inf'), (0,0), []

# Evaluate function at each random point

for i in range(num_points):

      x, y = x_random[i], y_random[i]

      z = multi_peak_function(x, y)

      all_points.append((x, y, z))

      if z > max_value:

            max_value = z

            max_coords = (x, y)

all_points.sort(key=lambda point: point[2], reverse=True)

print(all_points[0])

(2.03, 2.04) = 3.0299
```

Simulation

Detecting the Higgs Boson with simulations!

- Smashing bundles of protons together creates many different decay paths.
- CMS/Atlas Detectors can detect signals related to different decays
- How do you know when you've detected the signal of the Higgs Boson?
- Simulate every possible decay path and combination of decay paths with and without Higgs Boson predicted signal
- Compare simulated data to collected data!





Simulations as key part of frequentist statistics

Evaluating if treatments work

You're an alpaca shepherd and want to work out if a new shampoos increases the wool quality of your alpaca.

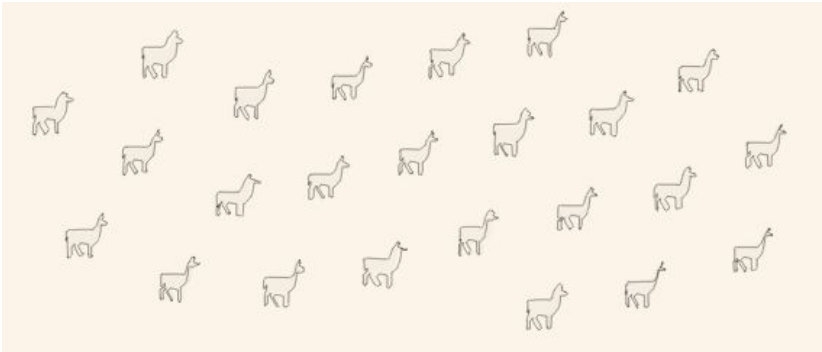
Our experiment will have the following hypotheses:

H_0: $\mu_{treatment} <= \mu_{control}$

Null: shampoo doesn't change average wool quality.

H_A: $\mu_{treatment} > \mu_{control}$

Alternative: new shampoo yields superior wool quality



https://www.jwilber.me/permutationtest/

Evaluating if treatments work

We can randomly assign our alpacas to:

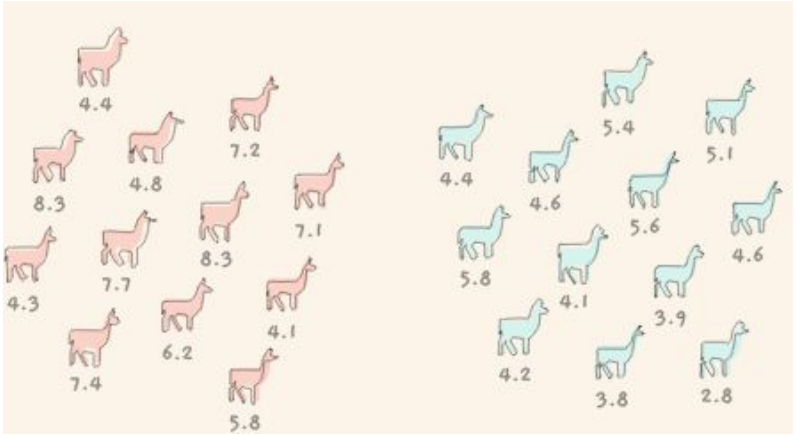
- Treatment group who gets the new shampoo
- Control group that keeps the old shampoo

Then we measure the wool quality in each alpaca and calculate the average for each group.

We can then calculate the difference in mean wool quality for each group to give us a test statistic:

Test Statistic = $\mu_{Treatment} - \mu_{Control}$

Test Statistic = 1.6




```
treat = np.array([4.8, 7.2, 6.2, …])
control = np.array([5.4, 4.4, 5.1, … ])

test_stat = np.mean(treat) - np.mean(control)
```

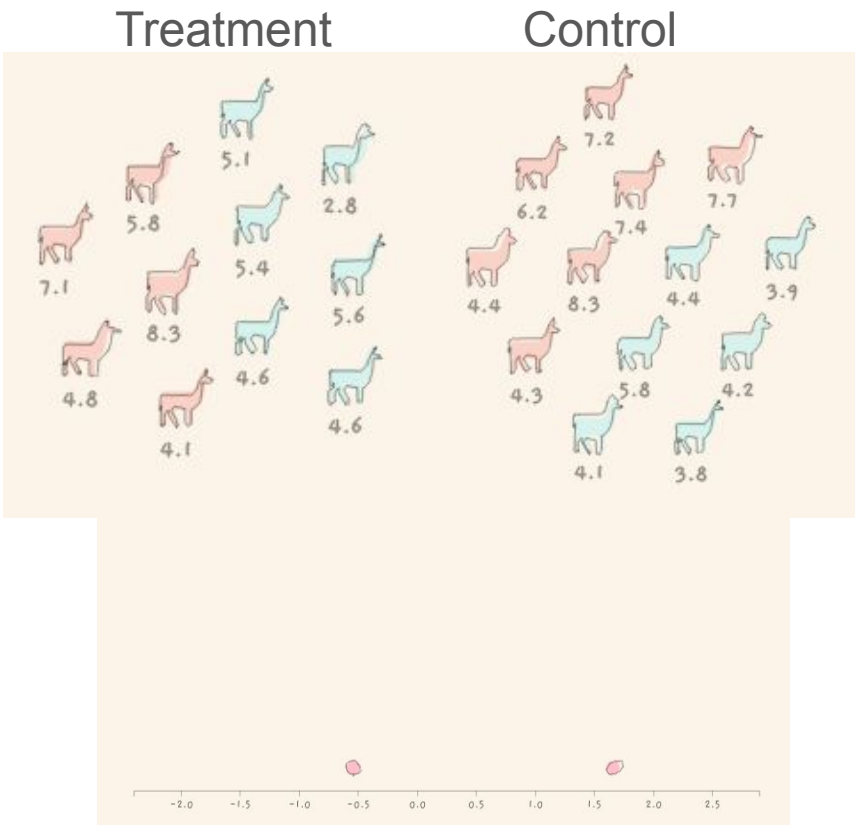
https://www.jwilber.me/permutationtest/

Determining whether a test statistic is significant

Need to know what the DISTRIBUTION of test statistics would look like if null were true

Can estimate this with simulation by shuffling the alpacas between groups and recalculating our test-statistic.

Shuffled test statistic = -0.5



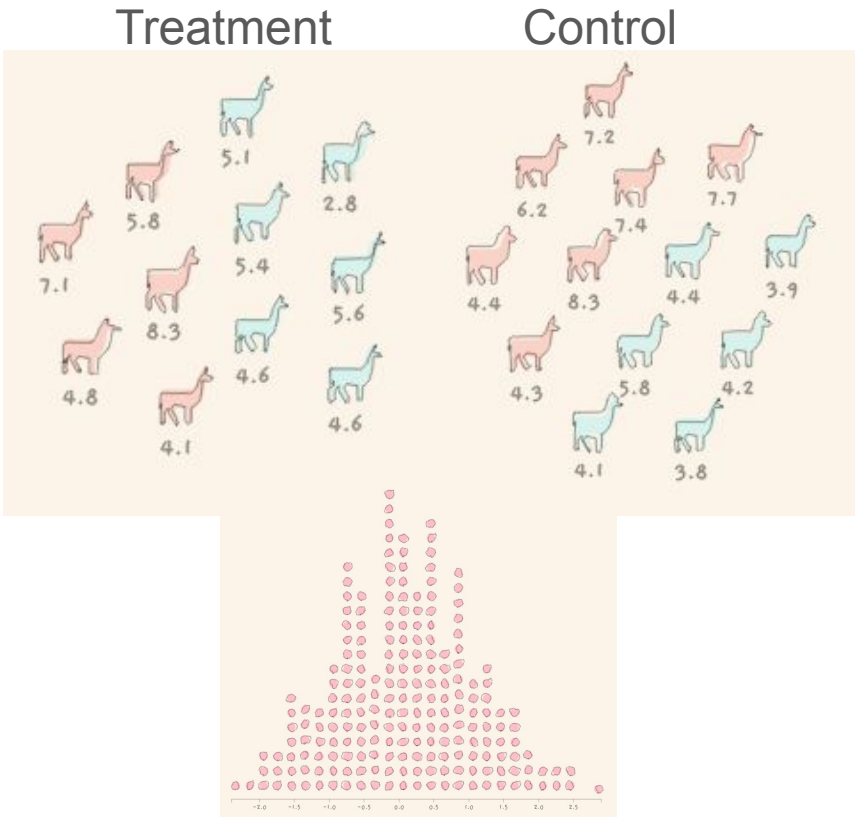
Determining whether a test statistic is significant

Need to know what the DISTRIBUTION of test statistics would look like if null were true

Can estimate this with simulation by shuffling the alpacas between groups and recalculating our test-statistic.

Shuffled test statistic = -0.5

Repeat many times = test distribution



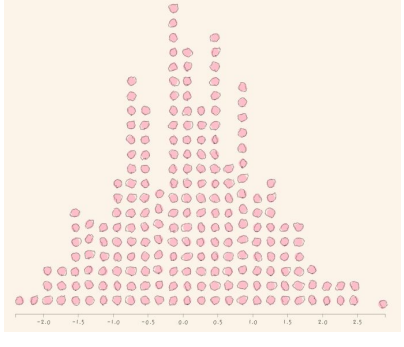
Determining whether a test statistic is significant

Need to know what the DISTRIBUTION of test statistics would look like if null were true

Can estimate this with simulation by shuffling the alpacas between groups and recalculating our test-statistic.

Shuffled test statistic = -0.5

Repeat many times = test distribution



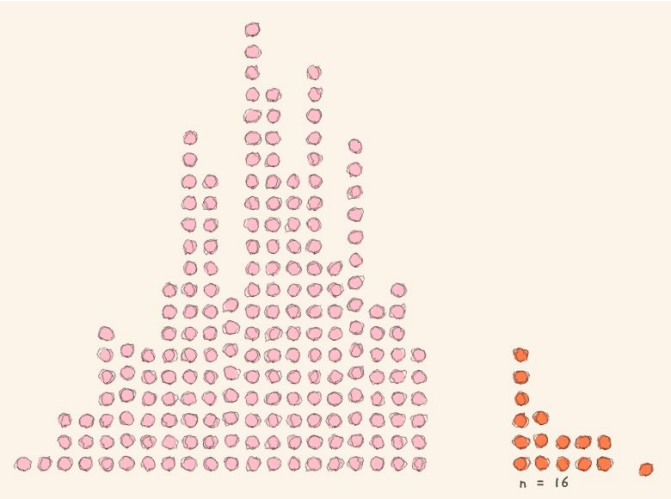
```
n_permutations = 200

combined = np.concatenate([treat, control])

n1, n2 = len(treat), len(control)

observed_diff = np.mean(treat) - np.mean(control)

null_distribution = np.zeros(n_permutations)

for i in range(n_permutations):

       np.random.shuffle(combined)

       new_treat = combined[:n1]

       new_control = combined[n1:n1+n2]

       null_distribution[i] = np.mean(new_tream) - np.mean(control)
```

Compare observed test statistics to null test-distribution

p-value is just the proportion of permutations where the sampled difference was greater than or equal to the difference we observed in our original treatment and control groups.

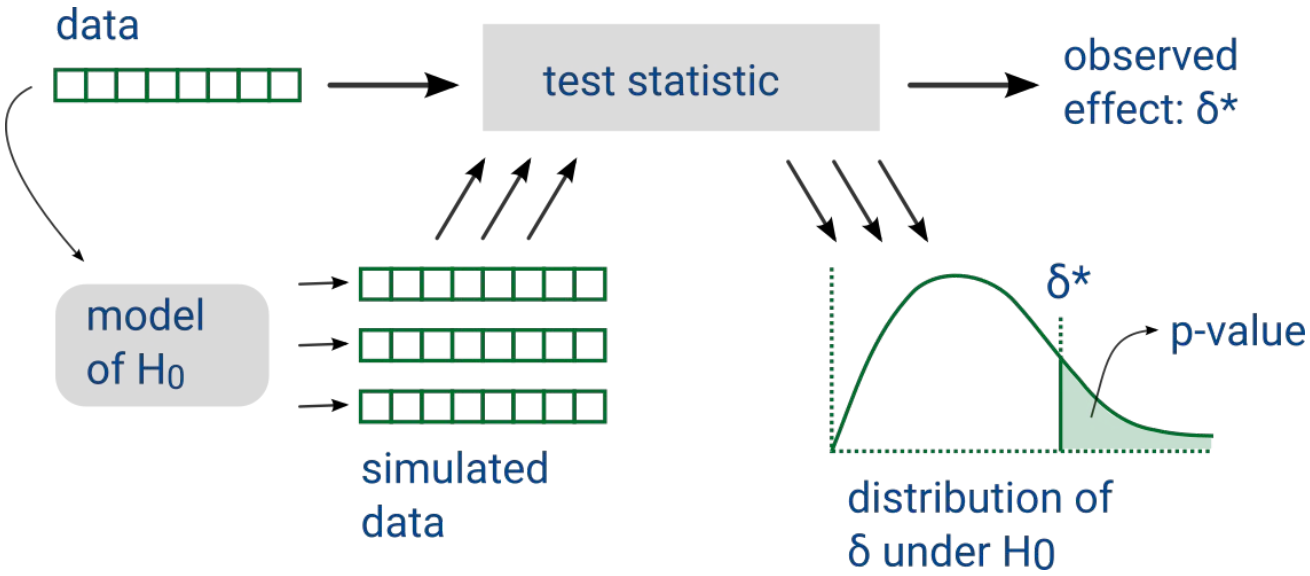
If we did 200 permutations and 16 had a difference >= to our observed test statistic

p-value = 16 / 200 = 0.08 or 8%




```
p_value = np.mean(np.abs(null_distribution) >= np.abs(test_statistic)
```

Frequent statistics really just boils down to this procedure!



You can memorise rules/common practices to do this analytically or more efficiently but essentially you are always comparing a test statistics to the distribution of that statistic under a null

Calculating confidence intervals by random re-sampling

Let's say we are measuring the average growth in a set of 15 plants.

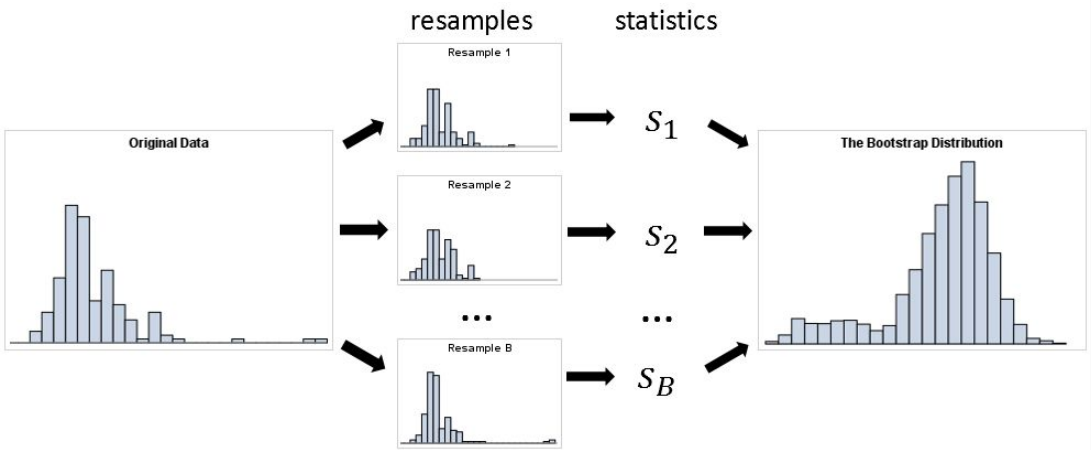
Can calculate the mean easily but how do we know what the plausible range of values would be.

A **confidence interval** gives a range of plausible values for an unknown population parameter (like a mean, proportion, or median)

Definition: a 95% confidence interval of [10.2, 12.8] means if you were to repeat your sampling process many times ~95% of the resulting intervals would contain the true population parameter.

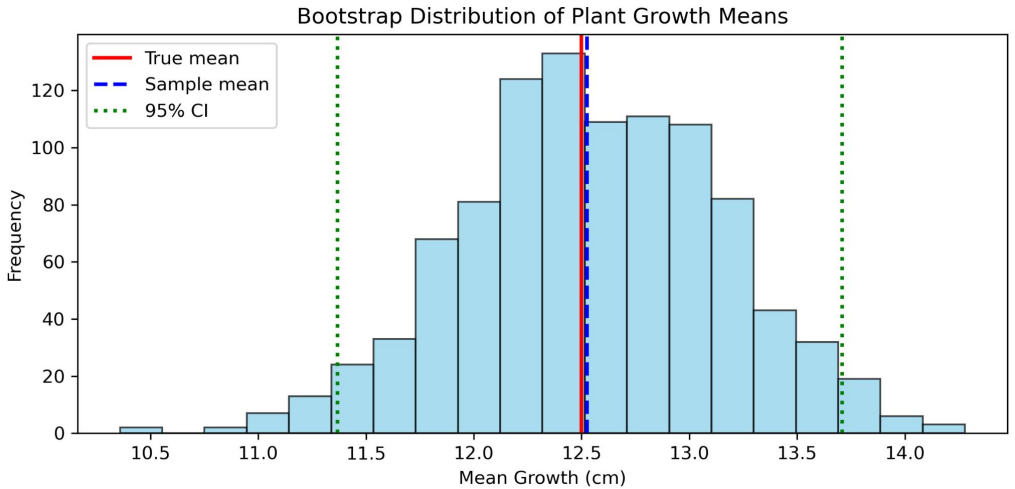
```
# We'll make up some data but in the real-world we'd
measure this in an experiment.

n_plants = 15

true_mean_growth = 3.25

growth_data = rng.normal(loc=true_mean_growth,

                         scale=2.5, size=n_plants)



# Calculate the sample mean

sample_mean = np.mean(growth_data)
```

Bootstrapping: sampling original data with replacement




```python
n_bootstrap = 1000

bootstrap_means = []

for _ in range(n_bootstrap):

    # Resample with replacement

    bootstrap_sample = np.random.choice(growth_data,

                                        size=len(growth_data),

                                        replace=True)

       bootstrap_means.append(np.mean(bootstrap_sample))

bootstrap_means = np.array(bootstrap_means)
```

Bootstrapping: sampling original data with replacement




```
# Calculate 95% confidence interval

lower_ci = np.percentile(bootstrap_means, 2.5)

upper_ci = np.percentile(bootstrap_means, 97.5)
```

Summary

- Numpy basics and gotcha
- Randomness
- Pseudorandom Number Generators
- Numpy PRNG functions
  - Using PRNGs to estimate values (Monte Carlo)
- Probability Distributions
  - Estimating outcomes by sampling from distributions
- Optimisation
  - Grid-search vs Random-search
- Simulation
  - Calculating p-values by simulating Null distribution
  - Bootstrapping to estimate confidence intervals