Python

- We will use Python (version 3.9+) and **`numpy`** (v1.23+) to simulate all our experiments.
    - Note that there are two types of NumPy random number generators, the "Legacy Random Generation" (v1.16 and earlier), and the new "Generator" (which is what we will use).
    - A lot of online documentation still uses the old one, which could confuse you.
    - Use the NumPy documentation as your textbook.
- If you don't know Python, please work through the introduction tutorial on SunLearn.
- You can run Python on your local PC or on Google Colab.

Generating Random Numbers

```
import numpy as np
from numpy.random import default_rng

rng = default_rng()
random_float = rng.random() # produces values [0,1)
print(random_float)
# OUTPUTS: 0.6744263927245717 (note this will be different for you)
```

Generating Random Numbers

Functions available in `default_rng()`

Simple random data

integers(low[, high, size, dtype, endpoint])	Return random integers from *low* (inclusive) to *high* (exclusive), or if endpoint=True, *low* (inclusive) to *high* (inclusive).
random([size, dtype, out])	Return random floats in the half-open interval [0.0, 1.0).
choice(a[, size, replace, p, axis, shuffle])	Generates a random sample from a given array
bytes(length)	Return random bytes.

Toss a coin

```
import numpy as np
from numpy.random import default_rng
rng = default_rng()

outcomes = ["heads", "tails"]

coin_toss_result = rng.choice(outcomes)
print(coin_toss_result)
# OUTPUTS: heads (note this will be different for you)
```

Roll some dice

Roll 10 dice, count the number of times that the number is greater than 4. What did you expect?

```
# normal imports here...

# roll 10 dices and store result as an array X
X = rng.integers(1, 6, 10, int, True)
print(X)
# OUTPUTS: [1 5 3 1 5 4 3 2 3 5] (note this will be different for you)
P_gt_4 = (X>4).sum() / len(X)
print(P_gt_4)
# OUTPUTS: 0.3 (note this will be different for you)
```

Now increase the number of dice and test the relative frequency formula: $P(A) = \lim_{n \to \infty} \frac{n_A}{n}$

Permutation and Combinations in Python

```python
from itertools import permutations, combinations

# Get all permutations of lenght 2
perm = permutations([1, 2, 3, 4], 2)
print("Permutations:", list(perm))

# Get all combinations of lenght 2
comb = combinations([1, 2, 3, 4], 2)
print("Combinations:", list(comb))
```

Outputs:

```
Permutations: [(1, 2), (1, 3), (1, 4), (2, 1), (2, 3),
(2, 4), (3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3)]

Combinations: [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(3, 4)]
```

Permutation and Combinations in Python

If you only need the actual number of permutations and combinations

```
import math
math.perm(n, k)
math.comb(n, k)
```

Python

PMFs and CDFs are often defined in as a piecewise function. Numpy has a convenient built-in function that can deal with these.

```
x = np.linspace(-2.5, 2.5, 6)
y = np.piecewise(x, [x < 0, x >= 0], [-1, 1])
# OUTPUT: array([-1., -1., -1.,  1.,  1.,  1.])
```

$$y = f(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

Families of Discrete Random Variables

You can of course generate these distributions manually, but **`scipy`** has many functions that help you generate data that follow the various distributions

- `scipy.stats.bernoulli`
- `scipy.stats.binom`
- `scipy.stats.geom`

Families of Discrete Random Variables

`scipy` can generate data that follow the various distributions

- `scipy.stats.nbinom`
  - For Pascal distribution, make $n := k$, and $x := x - k$
  - It's easier if you just use nbinom as normal, and then when you plot, you simply add `k` to `x`.
  - e.g. `plt.vlines((x+k), 0, pmf_y)`
- `scipy.stats.randint`
  - They use $x \in [k, l)$, so `low` $:= k$ and `high` $:= l + 1$
- `scipy.stats.poisson`
  - Let $\mu := \alpha$

Families of Continuous Random Variables

- Most software libraries that generate random numbers has some kind of `rand` function which produces numbers in the range $[0, 1)$.
  For NumPy: `numpy.random.default_rng().random()`
- `scipy.stats.uniform`
  - For uniform distribution, make `loc` $:= a$, and `scale` $:= b - a - 0.00001$ (since `scipy` generate it for $[0, 1]$)
- `scipy.stats.erlang`
  - make `scale` $:= 1/\lambda$, and `a` $:= n$
  - For exponential distribution, simply make `a` $:= 1$
  - See also `scipy.stats.gamma`