Module 2

Simulating memory recall

&

Learning Object Oriented Programming

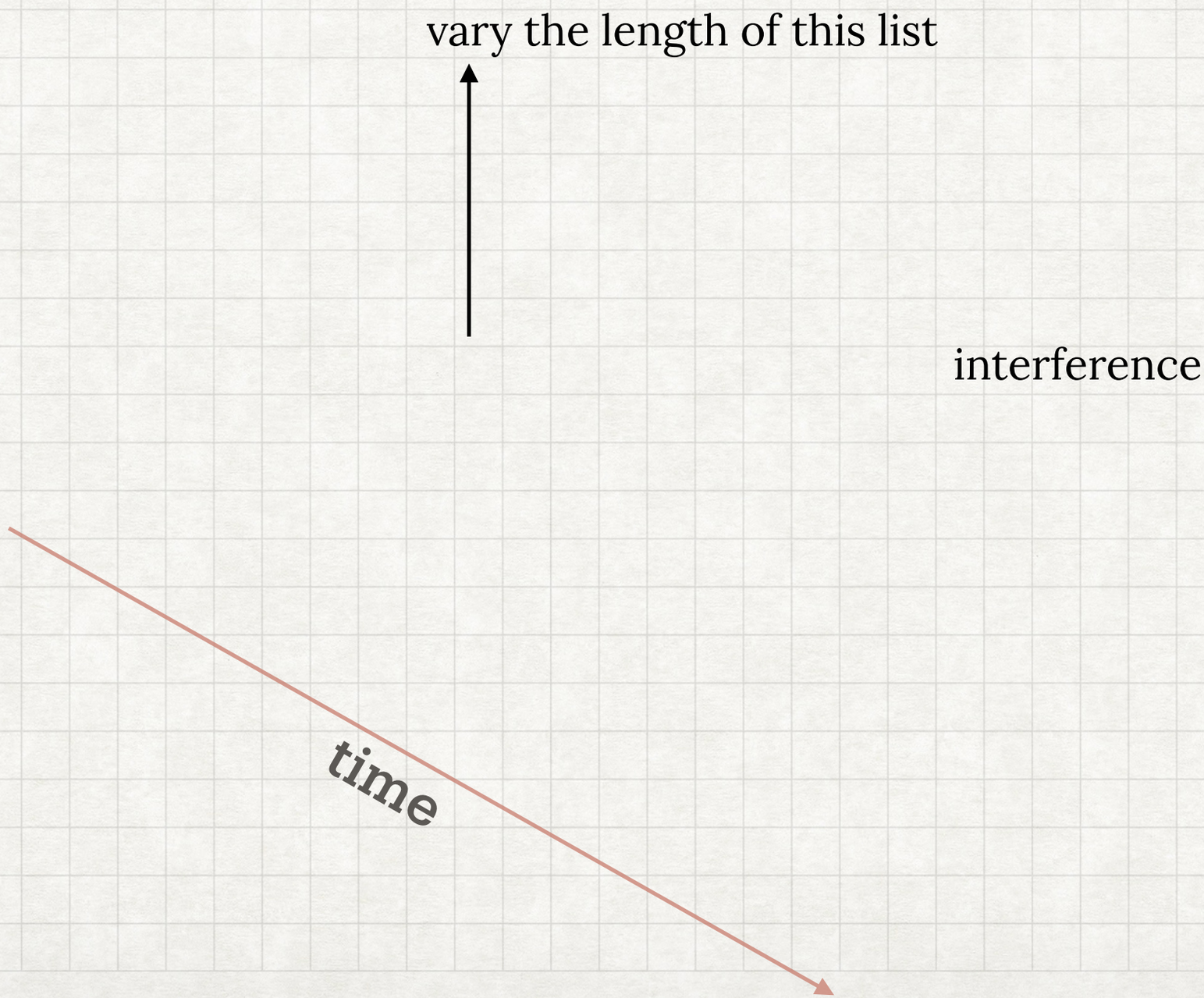
Resources:

https://realpython.com/python3-object-oriented-programming/

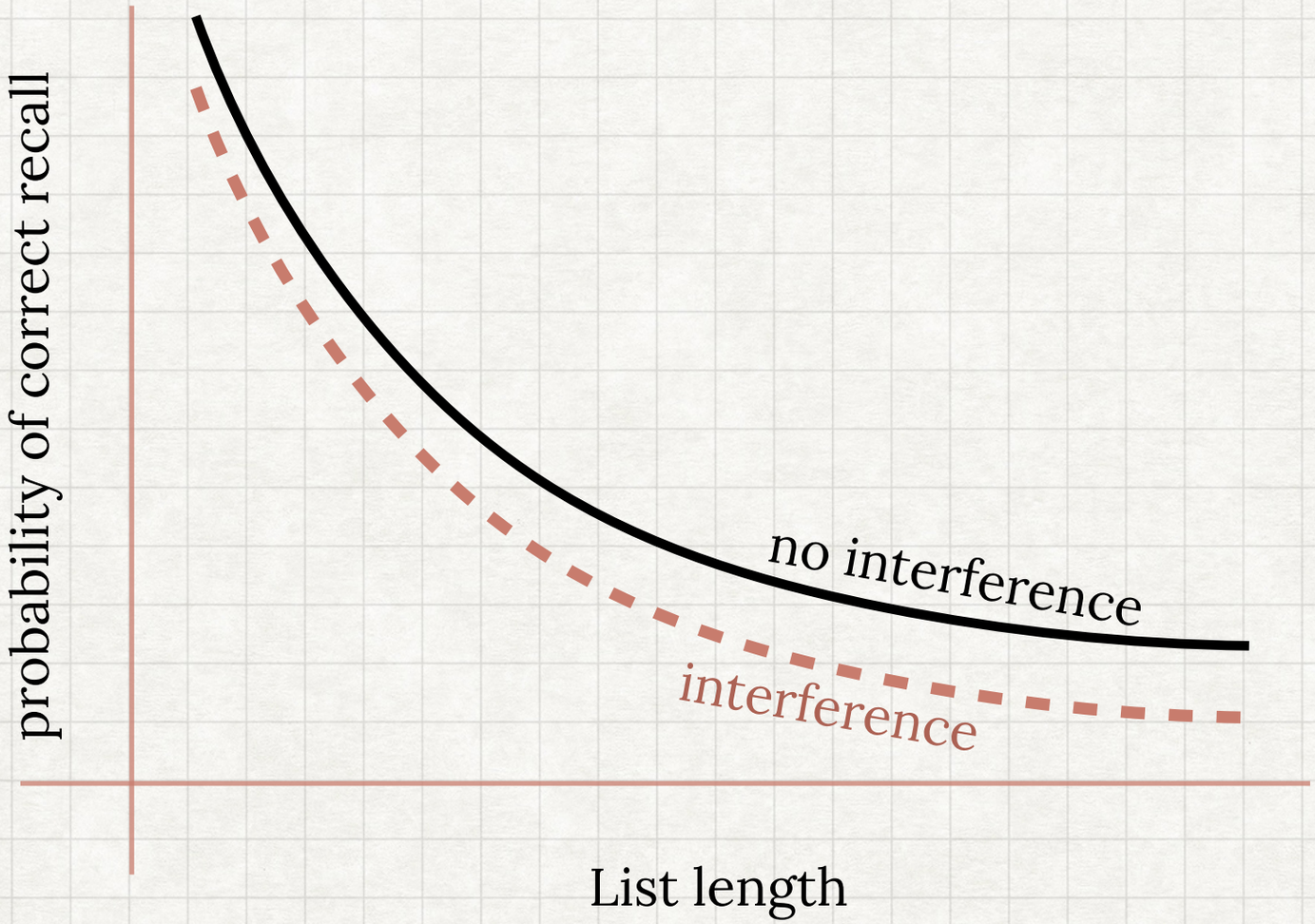
https://realpython.com/numpy-random-number-generator/

Mini-Project: List-length effect in memory

Recall a list of letters / syllables / non-words



List-length effect



Mini-project 2

Write Python code to:

* Simulate recall across list lengths, with and without interference, visualise the forgetting curves
* Use Object-oriented programming to:
  * Define a Baseline Model, which simulates forgetting based on list length
  * Define a Interference Model, which simulates effect of interference
* Run an experiment, where participants are divided into control & baseline conditions
* Simulate data from participants and store into CSV file
* Analyse the results and visualise the forgetting curves for both set of participants

Demo 1: Creating classes and instances

Write Python code to:

* Create a class, called ‘Coin’. This class:
    * Should contain two ‘sides’ (called ‘heads’ & ‘tails’)
    * Should contain a method called ‘flip’, that randomly returns one of the two ‘sides’

Demo 2: Creating a SimpleMemoryModel class

Write Python code to:

* Create a 'SimpleMemoryModel' class. This class:
    * Should have a mean recall rate, and a 'decay' in mean recall rate
    * Contain a method called 'simulateTrial' with argument 'list_length' where:
        * Every time the method is called, it simulates a recall from memory, with a probability determined by the mean recall rate and decay based on list_length

Demo 3: Inherit from a SimpleMemory class

Write Python code to:

* Create a 'InterferenceModel' class. This class:
    * Should do everything that the SimpleMemoryModel does
    * But impose an additional **penalty** on the recall probability during each simulate_trial() method

Demo 3: Inherit from a SimpleMemory class

Pseudocode:

```
DEFINE CLASS BaselineModel
    Like above…

DEFINE CLASS InterferenceModel EXTENDS BaselineModel
    METHOD __init__(penalty = 0.08, …parent's arguments…)
        CALL parent __init__ with parent's arguments
        STORE penalty
    METHOD simulate_trial(list_length)
        (p, acc) = CALL parent simulate_trial(list_length)
        p = p - penalty
        CLIP p to [0, 1]
        acc = 1 if rng.random() < p else 0   // resample after applying penalty
        RETURN (p, acc)

base = BaselineModel(seed = XX)
inter = InterferenceModel(seed = YY)
PRINT base.simulate_trial(5)
PRINT inter.simulate_trial(5)  // expect lower p
```

Demo 5: Modular programming

Divide the code into a set of files, in the following format:

* A file called models.py, contains:
    * SimpleMemoryModel and InterferenceModel
* A file called helpers.py, that contains:
    * A helper function to clip probabilities to 0 & 1.
* A file called main.py, that calls simulates a single trial in either the ‘baseline’ or ‘interference’ condition, based on command-line arguments.

Resources:

https://realpython.com/python-modules-packages/

Demo 5: Modular programming – Pseudocode

main.py

FUNCTION run(participants, repeats, out_files):

- Simulate group of participants → trial dataset.
- Save raw data to CSV.
- Plot forgetting curves and save figure. [[Leave this for Mini-project]]

MAIN PROGRAM:

- Parse command-line arguments (participants, repeats, output paths).
- Call run() with arguments.
- PRINT confirmation of outputs.

Mini Project 2: Forgetting curves

Flesh out our Python code to:

* Write a module simulate.py. This module should use models.py to simulate a participant for a given number of trials under one of the two conditions (Baseline, or Interference). Here is the pseudo-code:

FUNCTION simulate_participant(condition, list_lengths, repeats):

- Choose model type (baseline vs. interference).
- For each list length and repeat:
  - Run a trial, record probability & accuracy.
- RETURN table of trials for this participant.

* Add a function called 'simulate_group' that calls simulate_participant to simulate two groups of participants, one in each condition:

FUNCTION simulate_group(n_participants, list_lengths, repeats):

- For each participant:
  - Randomly assign condition.
  - Call simulate_participant().
  - Add participant ID & condition labels.
- RETURN combined dataset.

* Add a module for plotting results that contains a function called 'plot_results()' that takes a dataframe of results and generates forgetting curves.