This executable notebook will help you complete parts of Pset 5:

1. Scaffolding code for using regular expressions to syllabify words.

If you haven't used Colab before, it's very similar to Jupyter / IPython / R Notebooks: cells containing Python code can be interactively run, and their outputs will be interpolated into this document. If you haven't used any such software before, we recommend taking a quick tour of Colab.

Now, a few Colab-specific things to note about execution before we get started:

- Google offers free compute (including GPU compute!) on this notebook, but *only for a limited time*. Your session will be automatically closed after 12 hours. That means you'll want to finish within 12 hours of starting, or make sure to save your intermediate work (see the next bullet).
- You can save and write files from this notebook, but they are *not guaranteed to persist*. For this reason, we'll mount a Google Drive account and write to that Drive when any files need to be kept permanently (e.g. model checkpoints, surprisal data, etc.).
- You should keep this tab open until you're completely finished with the notebook. If you close the tab, your session will be marked as "Idle" and may be terminated.

Getting started

First, make a copy of this notebook so you can make your own changes. Click *File -> Save a copy in Drive*.

What you need to do

Read through this notebook and execute each cell in sequence, making modifications and adding code where necessary. You should execute all of the code as instructed, and make sure to write code or textual responses wherever the text **TODO** shows up in text and code cells.

When you're finished, choose *File -> Download .ipynb*. You will upload this `.ipynb` file as part of your submission.

Syllabification

Write a program that uses regular expressions to syllabify English words in their written orthographic form.

```
# import relevant libraries
import re
import pandas as pd
import numpy as np
```

```
def eval_syllabifications(examples,syllabifier):
  """
  Evaulates your syllabifier on the given examples
  """
  # easier test: did you get the right number of syllables?
  syllabifications = []
  count_correct = []
  syllabification_correct = []
  for example in examples:
    y = syllabify(example[0])
    syllabifications.append(y)
    count_correct.append(1 if len(y)==len(example[1]) else 0)
    syllabification_correct.append(1 if y==example[1] else 0)
  result = pd.DataFrame({'Example': [example[0] for example in
examples],
                         'Syllabification': syllabifications,
                         'CountCorrect':count_correct,

'SyllabificationCorrect':syllabification_correct})
  print("Syllable counts: " + str(sum(count_correct)) + " of " +
str(len(count_correct)))
  print("Syllabifications: " + str(sum(syllabification_correct)) + "
of " + str(len(syllabification_correct)))
  return(result)

examples = [('i',['i']), # note all lowercase
            ('air',['air']),
            ('big',['big']),
            ('strength',['strength']),
            ('steal',['steal']),
            ('ideal',['i','deal']),
            ('quiet',['qui','et']),
            ('enter',['en','ter']),
            ('able',['a','ble']),
            ('pandas',['pan','das']),
            ('intake',['in','take']),
            ('capable',['ca','pa','ble']),
            ('serendipity',['se','ren','di','pi','ty'])]

def eval_syllabifications(examples, syllabifier):
    """
    Evaluates your syllabifier on the given examples
    """
    # easier test: did you get the right number of syllables?
    syllabifications = []
    count_correct = []
    syllabification_correct = []

    for example in examples:
```

```
        y = syllabifier(example[0])
        syllabifications.append(y)
        count_correct.append(1 if len(y) == len(example[1]) else 0)
        syllabification_correct.append(1 if y == example[1] else 0)

    result = pd.DataFrame({
        'Example': [example[0] for example in examples],
        'Expected': [example[1] for example in examples],
        'Actual': syllabifications,
        'CountCorrect': count_correct,
        'SyllabificationCorrect': syllabification_correct
    })

    print("Syllable counts: " + str(sum(count_correct)) + " of " +
str(len(count_correct)))
    print("Syllabifications: " + str(sum(syllabification_correct)) + "
of " + str(len(syllabification_correct)))
    return result

def syllabify(word):
    """
    Syllabification for English orthographic words (including nonce
words)
    Uses general phonotactic rules rather than word-specific patterns
    """
    if not word:
        return []

    word = word.lower().strip()

    # Handle single character words
    if len(word) == 1:
        return [word]

    # Define vowel and consonant patterns
    vowels = 'aeiouy'
    consonants = 'bcdfghjklmnpqrstvwxz'

    # Check if word has any vowels - if not, it's one syllable
    if not any(c in vowels for c in word):
        return [word]

    # Step 1: Identify vowel nuclei (potential syllable centers)
    # Mark positions where vowels occur, considering diphthongs
    vowel_positions = []
    diphthongs = ['ai', 'au', 'ay', 'ea', 'ee', 'ei', 'eu', 'ey',
'ie', 'oa', 'oi', 'oo', 'ou', 'oy', 'ue', 'ui', 'aw', 'ew', 'ow']

    i = 0
    while i < len(word):
```

```
        if word[i] in vowels:
            # Check for diphthong
            if i < len(word) - 1 and word[i:i+2] in diphthongs:
                vowel_positions.append((i, i+1))  # diphthong spans
two positions
                i += 2
            else:
                vowel_positions.append((i, i))    # single vowel
                i += 1
        else:
            i += 1

    # If only one vowel nucleus, it's one syllable
    if len(vowel_positions) <= 1:
        return [word]

    # Step 2: Determine syllable boundaries based on consonant
patterns
    syllable_boundaries = [0]  # Start of word

    for j in range(len(vowel_positions) - 1):
        v1_end = vowel_positions[j][1]      # End of current vowel
        v2_start = vowel_positions[j+1][0]  # Start of next vowel

        # Find consonants between vowels
        consonant_cluster = word[v1_end + 1:v2_start]

        if not consonant_cluster:
            # Adjacent vowels - split between them (already handled by
vowel detection)
            syllable_boundaries.append(v1_end + 1)
        elif len(consonant_cluster) == 1:
            # Single consonant: goes with following vowel (CV
preference)
            syllable_boundaries.append(v1_end + 1)
        else:
            # Multiple consonants: apply sonority and onset
maximization
            # Split to maximize legal onsets for the following
syllable

            # Comprehensive English onset clusters (beginnings of
syllables)
            legal_onsets = {
                # Single consonants (all are legal)
                'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n',
'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'z',
                # Two-consonant clusters
                'bl', 'br', 'cl', 'cr', 'dr', 'fl', 'fr', 'gl', 'gr',
'pl', 'pr', 'tr', 'tw',
```

```
                'sc', 'sk', 'sl', 'sm', 'sn', 'sp', 'st', 'sw', 'sh',
'th', 'wh', 'ch',
                'dw', 'gw', 'kw', 'pw', 'sw', 'tw', 'qu',
                # Three-consonant clusters
                'scr', 'spl', 'spr', 'str', 'squ', 'thr', 'shr', 'chr'
            }

            # Illegal consonant combinations that should never start a
syllable
            illegal_onsets = {
                'tl', 'dl', 'tn', 'dn', 'tm', 'dm', 'tp', 'dp', 'tk',
'dk', 'tb', 'db',
                'gn', 'kn', 'pn', 'mn', 'ng', 'nk', 'nt', 'nd', 'mp',
'mb', 'lk', 'rk',
                'ls', 'rs', 'ts', 'ds', 'ps', 'ks', 'fs', 'hs'
            }

            # Try to find the longest legal onset, avoiding illegal
combinations
            split_point = v1_end + len(consonant_cluster)  # Default:
all consonants with first vowel

            # Work backwards from the longest possible onset
            for onset_len in range(min(3, len(consonant_cluster)), 0,
-1):
                potential_onset = consonant_cluster[-onset_len:]

                # Check if this onset is legal
                if (potential_onset in legal_onsets and
                    potential_onset not in illegal_onsets):
                    split_point = v2_start - onset_len
                    break

            # Special case: if we couldn't find any legal onset,
            # just take the last consonant (most consonants can start
syllables alone)
            if split_point == v1_end + len(consonant_cluster) and
len(consonant_cluster) > 0:
                last_consonant = consonant_cluster[-1]
                if last_consonant not in 'xq':  # Almost all single
consonants work except a few
                    split_point = v2_start - 1

            syllable_boundaries.append(split_point)

    syllable_boundaries.append(len(word))  # End of word

    # Step 3: Extract syllables based on boundaries
    syllables = []
    for k in range(len(syllable_boundaries) - 1):
```

```
        start = syllable_boundaries[k]
        end = syllable_boundaries[k + 1]
        syllable = word[start:end]
        if syllable:  # Make sure syllable is not empty
            syllables.append(syllable)

    # Step 4: Post-processing - ensure each syllable has a vowel
    final_syllables = []
    i = 0
    while i < len(syllables):
        current = syllables[i]

        # If current syllable has no vowel, merge with adjacent
syllable
        if not any(v in current for v in vowels):
            if i > 0:  # Merge with previous
                final_syllables[-1] += current
            elif i < len(syllables) - 1:  # Merge with next
                syllables[i + 1] = current + syllables[i + 1]
            else:  # Single consonant-only word
                final_syllables.append(current)
        else:
            final_syllables.append(current)
        i += 1

    # Ensure we return at least the original word if something went
wrong
    if not final_syllables:
        return [word]

    return final_syllables

eval_syllabifications(examples,syllabify)
```

```
Syllable counts: 12 of 13
Syllabifications: 12 of 13
```


```
{"summary":"{\n  \"name\":
\"eval_syllabifications(examples,syllabify)\",\n  \"rows\": 13,\n
\"fields\": [\n    {\n      \"column\": \"Example\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 13,\n        \"samples\": [\n
\"capable\",\n          \"pandas\",\n          \"i\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Expected\",\n      \"properties\":
{\n        \"dtype\": \"object\",\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Actual\",\n      \"properties\": {\n        \"dtype\": \"object\",\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"CountCorrect\",\n
```

```
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0,\n        \"min\": 0,\n        \"max\": 1,\n
\"num_unique_values\": 2,\n        \"samples\": [\n          0,\n
1\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"SyllabificationCorrect\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0,\n        \"min\": 0,\n
\"max\": 1,\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          0,\n          1\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe"}
```


Your function should also handle *nonce* English orthographic words---letter sequences that don't happen to be words, but that could be. For example:

```
nonce_example = [('sneed',['sneed']),
                 ('snoded',['sno','ded']),
                 ('ilskip',['il','skip'])]

eval_syllabifications(nonce_example,syllabify)
```

```
Syllable counts: 3 of 3
Syllabifications: 3 of 3
```


```
{"summary":"{\n  \"name\":
\"eval_syllabifications(nonce_example,syllabify)\",\n  \"rows\": 3,\n
\"fields\": [\n    {\n      \"column\": \"Example\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 3,\n        \"samples\": [\n
\"sneed\",\n          \"snoded\",\n          \"ilskip\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Expected\",\n      \"properties\":
{\n        \"dtype\": \"object\",\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Actual\",\n      \"properties\": {\n        \"dtype\": \"object\",\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"CountCorrect\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0,\n        \"min\": 1,\n        \"max\": 1,\n
\"num_unique_values\": 1,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"SyllabificationCorrect\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0,\n        \"min\": 1,\n        \"max\": 1,\n
\"num_unique_values\": 1,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    }\n  ]\n}","type":"dataframe"}
```


For cases your program is not able to capture from the above examples, provide an explanation of why you think that is the case.

TODO: The code is failing on the word "intake" since as the word is broken down vowel by vowel instead of recognizing it as a compound word. Here's what happens: The algorithm finds three vowels - 'i', 'a', and 'e' - and tries to split the consonants between them. It sees the "nt" cluster between 'i' and 'a', splits it so 'n' stays with 'i' and 't' goes with 'a'. Then it sees 'k' between 'a' and 'e' and puts it with 'e'. This gives us "in-ta-ke". But "intake" is really the compound word "in" + "take", where "take" should stay as one syllable. The algorithm doesn't know about word boundaries - it just follows phonetic splitting rules. So it treats "take" like any other sequence of sounds rather than recognizing it as a meaningful unit that shouldn't be broken up.

Further, try your implementation on new English

> Add blockquote

words of your choosing, and describe what kinds of words you find hard to handle. Make sure to include examples where the program succeeds, and examples where it fails.

TODO:

```
# Testing on additional words to find strengths and weaknesses
print("\n" + "="*60)
print("TESTING ON DIVERSE ENGLISH WORDS")
print("="*60)

# Words that should work well (regular phonetic patterns)
easy_words = [
    ('computer', ['com', 'pu', 'ter']),
    ('elephant', ['el', 'e', 'phant']),  # or ['el', 'ephant']
    ('fantastic', ['fan', 'tas', 'tic']),
    ('deliver', ['de', 'liv', 'er']),
    ('banana', ['ba', 'na', 'na']),
    ('telephone', ['tel', 'e', 'phone']),
]

print("WORDS THAT SHOULD WORK WELL (regular patterns):")
easy_result = eval_syllabifications(easy_words, syllabify)
print(easy_result[['Example', 'Expected', 'Actual',
'SyllabificationCorrect']].to_string(index=False))

# Compound words (likely to have issues)
compound_words = [
    ('playground', ['play', 'ground']),
    ('birthday', ['birth', 'day']),
    ('sunshine', ['sun', 'shine']),
    ('football', ['foot', 'ball']),
    ('backyard', ['back', 'yard']),
    ('outside', ['out', 'side']),
    ('ghosting', ['ghost', 'ing']),
    ('Doomscrolling', ['Doom', 'scrol', 'ling'])
]

print(f"\nCOMPOUND WORDS (challenging - morphology vs phonetics):")
```

```python
compound_result = eval_syllabifications(compound_words, syllabify)
print(compound_result[['Example', 'Expected', 'Actual',
'SyllabificationCorrect']].to_string(index=False))

# Words with silent letters or irregular patterns
irregular_words = [
    ('castle', ['cas', 'tle']),  # silent 't'
    ('listen', ['lis', 'ten']),  # silent 't'
    ('island', ['is', 'land']),  # silent 's'
    ('knife', ['knife']),        # silent 'k'
    ('knee', ['knee']),          # silent 'k'
    ('honest', ['hon', 'est']),  # silent 'h'
]

print(f"\nWORDS WITH SILENT LETTERS (very challenging):")
irregular_result = eval_syllabifications(irregular_words, syllabify)
print(irregular_result[['Example', 'Expected', 'Actual',
'SyllabificationCorrect']].to_string(index=False))

# Words with unusual consonant clusters
cluster_words = [
    ('rhythm', ['rhythm']),      # no vowels except 'y'
    ('sixth', ['sixth']),        # complex ending
    ('twelfth', ['twelfth']),    # complex cluster
    ('strengths', ['strengths']), # very complex
    ('scratched', ['scratched']), # or ['scratch', 'ed']
    ('glimpse', ['glimpse']),
]

print(f"\nCOMPLEX CONSONANT CLUSTERS:")
cluster_result = eval_syllabifications(cluster_words, syllabify)
print(cluster_result[['Example', 'Expected', 'Actual',
'SyllabificationCorrect']].to_string(index=False))
```

```
============================================================
TESTING ON DIVERSE ENGLISH WORDS
============================================================
WORDS THAT SHOULD WORK WELL (regular patterns):
Syllable counts: 5 of 6
Syllabifications: 2 of 6
  Example        Expected            Actual  SyllabificationCorrect
 computer  [com, pu, ter]    [com, pu, ter]                       1
 elephant  [el, e, phant]    [e, lep, hant]                       0
fantastic [fan, tas, tic]   [fan, ta, stic]                       0
  deliver   [de, liv, er]     [de, li, ver]                       0
   banana    [ba, na, na]      [ba, na, na]                       1
telephone [tel, e, phone] [te, lep, ho, ne]                       0

COMPOUND WORDS (challenging - morphology vs phonetics):
```

```
Syllable counts: 5 of 8
Syllabifications: 3 of 8
      Example            Expected              Actual
SyllabificationCorrect
   playground      [play, ground]      [play, ground]
1
     birthday        [birth, day]        [birth, day]
1
     sunshine        [sun, shine]      [sun, shi, ne]
0
     football        [foot, ball]        [foot, ball]
1
     backyard        [back, yard]      [bac, ky, ard]
0
      outside         [out, side]       [out, si, de]
0
     ghosting        [ghost, ing]        [gho, sting]
0
Doomscrolling [Doom, scrol, ling] [doom, scrol, ling]
0

WORDS WITH SILENT LETTERS (very challenging):
Syllable counts: 5 of 6
Syllabifications: 1 of 6
Example   Expected     Actual  SyllabificationCorrect
 castle [cas, tle] [cast, le]                       0
 listen [lis, ten] [li, sten]                       0
 island [is, land] [i, sland]                       0
  knife    [knife]  [kni, fe]                       0
   knee     [knee]     [knee]                       1
 honest [hon, est] [ho, nest]                       0

COMPLEX CONSONANT CLUSTERS:
Syllable counts: 4 of 6
Syllabifications: 4 of 6
  Example    Expected        Actual  SyllabificationCorrect
   rhythm    [rhythm]      [rhythm]                       1
    sixth     [sixth]       [sixth]                       1
  twelfth   [twelfth]     [twelfth]                       1
strengths [strengths]   [strengths]                       1
scratched [scratched] [scrat, ched]                       0
  glimpse   [glimpse]   [glimp, se]                       0
```

Summary: The code struggles most with:

Vowel cluster handling - Incorrectly splits adjacent/near-adjacent vowels (elephant → e-lep-hant) Consonant cluster boundaries - Inconsistent rules for splitting clusters between vowels (fantastic → fan-ta-stic) Silent letters - Treats all written consonants as pronounced (castle → cast-le) Complex word endings - Over-splits words that should be single syllables (glimpse → glimp-se)

The main weakness is inconsistent application of phonotactic rules for vowel detection and consonant splitting, not just morphological awareness.