Morphology_101_Workshop

September 12, 2025

1 Introduction to Morphology : Hands-on Experience

Goal: Understand the basics of morphology and try simple morphological analysis in Python.

1.1 What is Morphology?

Morphology is the branch of linguistics that studies the **internal structure of words** and how words are built from smaller meaningful units called **morphemes**.

> **Morpheme** = the smallest unit of meaning.
> Example: **cats** = *cat* (root) + *-s* (plural inflection).

Why it matters (NLP & AI): - Better search and retrieval (grouping word variants) - Lemmatization and stemming for preprocessing - Spell checking, machine translation, speech, grammar tools

1.1.1 Affix-stripping algorithms in Python.

These are stemmers that remove suffixes/prefixes using rules. The most common examples are Porter Stemmer, Lancaster Stemmer, and Snowball Stemmer, all available in NLTK.

[6]:
```
# Install nltk if not already installed
# pip install nltk

import nltk
from nltk.stem import PorterStemmer, LancasterStemmer, SnowballStemmer

# Initialize stemmers
porter = PorterStemmer()
lancaster = LancasterStemmer()
snowball = SnowballStemmer("english")

# Sample words
words = ["running", "happiness", "studies", "leaves", "better", "caresses",␣
 ↪"fishing"]

print("Word\tPorter\tLancaster\tSnowball")
print("-"*50)
for w in words:
```

```
    print(f"{w:10} {porter.stem(w):10} {lancaster.stem(w):10} {snowball.stem(w):
↪10}")
```

```
Word       Porter     Lancaster  Snowball
-------------------------------------------------
running    run        run        run
happiness  happi      happy      happi
studies    studi      study      studi
leaves     leav       leav       leav
better     better     bet        better
caresses   caress     caress     caress
fishing    fish       fish       fish
```

1.1.2 Hands-on 01: Minimal Affix-Stripping Analyzer

Below we implement a tiny **rule-based** analyzer to demonstrate: - Removing common English suffixes: `-s/-es/-ies` (plural / 3rd sg), `-ed` (past), `-ing` (progressive), `-er/-est` (comparative/superlative), `-ly` (adverb). - A few **spelling rules**: consonant undoubling (`stopped → stop`), restore dropped `e` (`hoped → hope`), and `y  i` alternation (`tries → try`, `happiest → happy`).

[13]:
```
# pip install nltk

import nltk
from nltk import pos_tag, word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import wordnet as wn

# ---- one-time downloads (uncomment the first time you run) ----
# nltk.download('punkt')
# nltk.download('averaged_perceptron_tagger')        # older NLTK
# nltk.download('averaged_perceptron_tagger_eng')    # newer NLTK (if needed)
# nltk.download('wordnet')
# nltk.download('omw-1.4')

def penn_to_wn(tag: str):
    """Map Penn POS to WordNet POS for better lemmatization."""
    if not tag:
        return wn.NOUN
    t = tag[0].upper()
    return {'J': wn.ADJ, 'V': wn.VERB, 'N': wn.NOUN, 'R': wn.ADV}.get(t, wn.
↪NOUN)

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

sentence = "The cats were running joyfully and chased the mice."
tokens = word_tokenize(sentence)
```

```
# Try new tagger name first; fall back if not present
try:
    tagged_tokens = pos_tag(tokens, lang='eng')
except LookupError:
    nltk.download('averaged_perceptron_tagger_eng')
    tagged_tokens = pos_tag(tokens, lang='eng')

print(f"{'Word':<12} {'POS':<10} {'Stem':<12} {'Lemma':<12}")
print("-" * 50)

for word, tag in tagged_tokens:
    stem = stemmer.stem(word)
    wn_pos = penn_to_wn(tag)
    lemma = lemmatizer.lemmatize(word, pos=wn_pos)
    print(f"{word:<12} {tag:<10} {stem:<12} {lemma:<12}")
```

```
Word         POS        Stem         Lemma
--------------------------------------------------
The          DT         the          The
cats         NNS        cat          cat
were         VBD        were         be
running      VBG        run          run
joyfully     RB         joy          joyfully
and          CC         and          and
chased       VBD        chase        chase
the          DT         the          the
mice         NN         mice         mouse
.            .          .            .
```

1.1.3 Basic morphological analyzer in Python using NLTK

[14]:
```
# pip install nltk

import nltk
# ---- one-time downloads (uncomment the first time) ----
# nltk.download('punkt')
# nltk.download('averaged_perceptron_tagger')
# nltk.download('averaged_perceptron_tagger_eng')  # for newer NLTK versions
# nltk.download('wordnet')
# nltk.download('omw-1.4')

from nltk import word_tokenize, pos_tag
from nltk.stem import PorterStemmer, SnowballStemmer, LancasterStemmer,␣
 ↪WordNetLemmatizer
from nltk.corpus import wordnet as wn

porter = PorterStemmer()
```

```
snowball = SnowballStemmer("english")
lancaster = LancasterStemmer()
lemmatizer = WordNetLemmatizer()

# Map Penn Treebank POS tags to WordNet POS
def penn_to_wn(tag: str):
    if not tag:
        return wn.NOUN
    t = tag[0].upper()
    if t == 'J':  # Adjective
        return wn.ADJ
    if t == 'V':  # Verb
        return wn.VERB
    if t == 'N':  # Noun
        return wn.NOUN
    if t == 'R':  # Adverb
        return wn.ADV
    return wn.NOUN

def analyze_text(text: str):
    tokens = word_tokenize(text)
    # Newer NLTK uses 'averaged_perceptron_tagger_eng'; fall back to old if␣
 ↪needed
    try:
        tagged = pos_tag(tokens, tagset=None, lang='eng')
    except LookupError:
        nltk.download('averaged_perceptron_tagger_eng')
        tagged = pos_tag(tokens, tagset=None, lang='eng')

    rows = []
    for tok, tag in tagged:
        wn_pos = penn_to_wn(tag)
        lemma = lemmatizer.lemmatize(tok, pos=wn_pos)
        rows.append({
            "token": tok,
            "pos": tag,
            "lemma": lemma,
            "porter": porter.stem(tok),
            "snowball": snowball.stem(tok),
            "lancaster": lancaster.stem(tok),
        })
    return rows

# --- Demo ---
text = "The children were running faster than the happiest runners and studied␣
 ↪boxes thoughtfully."
for row in analyze_text(text):
```

```
    print(f"{row['token']:12s} POS={row['pos']:6s}  lemma={row['lemma']:10s}  "
          f"porter={row['porter']:10s} snowball={row['snowball']:10s}␣
 ↪lancaster={row['lancaster']:10s}")
```

```
The          POS=DT     lemma=The         porter=the        snowball=the
lancaster=the
children     POS=NNS    lemma=child       porter=children   snowball=children
lancaster=childr
were         POS=VBD    lemma=be          porter=were       snowball=were
lancaster=wer
running      POS=VBG    lemma=run         porter=run        snowball=run
lancaster=run
faster       POS=RBR    lemma=faster      porter=faster     snowball=faster
lancaster=fast
than         POS=IN     lemma=than        porter=than       snowball=than
lancaster=than
the          POS=DT     lemma=the         porter=the        snowball=the
lancaster=the
happiest     POS=JJS    lemma=happy       porter=happiest   snowball=happiest
lancaster=happiest
runners      POS=NNS    lemma=runner      porter=runner     snowball=runner
lancaster=run
and          POS=CC     lemma=and         porter=and        snowball=and
lancaster=and
studied      POS=VBD    lemma=study       porter=studi      snowball=studi
lancaster=study
boxes        POS=NNS    lemma=box         porter=box        snowball=box
lancaster=box
thoughtfully POS=RB     lemma=thoughtfully  porter=thought    snowball=thought
lancaster=thought
.            POS=.      lemma=.           porter=.          snowball=.
lancaster=.
```

[1]:
```
# Minimal English morphological analyzer (toy, rule-based)
VOWELS = set("aeiou")
PREFIXES = ["un", "re", "dis", "non", "pre", "mis", "over", "under"]

def strip_prefix(w):
    for p in sorted(PREFIXES, key=len, reverse=True):
        if w.startswith(p) and len(w) > len(p) + 2:
            return p, w[len(p):]
    return "", w

def undouble_consonant(stem):
    return stem[:-1] if len(stem) >= 2 and stem[-1] == stem[-2] and stem[-1]␣
 ↪not in VOWELS else stem
```

```
def analyze(word):
    w = word.lower()
    result = {
        "token": word,
        "prefix": "",
        "stem": w,
        "suffix": "",
        "pos_guess": "UNK",
        "features": {}
    }

    # try prefix
    pfx, core = strip_prefix(w)
    if pfx:
        result["prefix"] = pfx
    else:
        core = w

    # small irregulars (examples only)
    irregulars = {
        "children": ("child", {"pos": "NOUN", "number": "PLUR"}),
        "men": ("man", {"pos": "NOUN", "number": "PLUR"}),
        "women": ("woman", {"pos": "NOUN", "number": "PLUR"}),
        "went": ("go", {"pos": "VERB", "tense": "PAST"}),
        "saw": ("see", {"pos": "VERB", "tense": "PAST"}),
        "better": ("good", {"pos": "ADJ", "degree": "COMP"}),
        "best": ("good", {"pos": "ADJ", "degree": "SUPER"})
    }
    if core in irregulars:
        lemma, feats = irregulars[core]
        result.update({"stem": lemma, "suffix": "", "pos_guess": feats.
↪pop("pos")})
        result["features"] = feats
        return result

    # adverbs -ly
    if core.endswith("ly") and len(core) > 3:
        result["stem"] = core[:-2]
        result["suffix"] = "ly"
        result["pos_guess"] = "ADV"
        return result

    # adjectives comparative/superlative -er/-est
    if core.endswith("est") and len(core) > 4:
        stem = core[:-3]
        stem = undouble_consonant(stem)
        if not stem.endswith("e"): stem += "e"  # nice->nicest->nice
```

```
        result.update({"stem": stem, "suffix": "est", "pos_guess": "ADJ",␣
↪"features": {"degree": "SUPER"}})
        return result
    if core.endswith("er") and len(core) > 3:
        stem = core[:-2]
        stem = undouble_consonant(stem)
        if not stem.endswith("e"): stem += "e"
        result.update({"stem": stem, "suffix": "er", "pos_guess": "ADJ",␣
↪"features": {"degree": "COMP"}})
        return result

    # verbs: -ing, -ed, 3sg -s/-es/-ies
    if core.endswith("ing") and len(core) > 4:
        stem = core[:-3]
        if stem and stem[-1] not in VOWELS and len(stem) >= 2 and stem[-1] ==␣
↪stem[-2]:
            stem = stem[:-1]          # running -> run
        elif not stem.endswith("e"):
            stem += "e"               # hoping -> hope
        result.update({"stem": stem, "suffix": "ing", "pos_guess": "VERB",␣
↪"features": {"aspect": "PROG"}})
        return result

    if core.endswith("ied") and len(core) > 3:
        result.update({"stem": core[:-3] + "y", "suffix": "ied", "pos_guess":␣
↪"VERB", "features": {"tense": "PAST/PART"}})
        return result
    if core.endswith("ed") and len(core) > 3:
        stem = core[:-2]
        if stem and stem[-1] not in VOWELS and len(stem) >= 2 and stem[-1] ==␣
↪stem[-2]:
            stem = stem[:-1]          # stopped -> stop
        elif not stem.endswith("e"):
            stem += "e"               # hoped -> hope
        result.update({"stem": stem, "suffix": "ed", "pos_guess": "VERB",␣
↪"features": {"tense": "PAST/PART"}})
        return result

    # present 3sg endings (similar to plural)
    if core.endswith(("ches", "shes", "xes", "zes", "ses")):
        result.update({"stem": core[:-2], "suffix": "es", "pos_guess": "VERB",␣
↪"features": {"tense": "PRES.3SG"}})
        return result
    if core.endswith("ies") and len(core) > 3:
        result.update({"stem": core[:-3] + "y", "suffix": "ies", "pos_guess":␣
↪"VERB", "features": {"tense": "PRES.3SG"}})
```

```
        return result
    if core.endswith("s") and not core.endswith("ss"):
        result.update({"stem": core[:-1], "suffix": "s", "pos_guess": "VERB",␣
↪"features": {"tense": "PRES.3SG"}})
        return result

    # nouns plural (fallback)
    if core.endswith(("ches", "shes", "xes", "zes", "ses")):
        result.update({"stem": core[:-2], "suffix": "es", "pos_guess": "NOUN",␣
↪"features": {"number": "PLUR"}})
        return result
    if core.endswith("ies") and len(core) > 3:
        result.update({"stem": core[:-3] + "y", "suffix": "ies", "pos_guess":␣
↪"NOUN", "features": {"number": "PLUR"}})
        return result
    if core.endswith("s") and not core.endswith("ss"):
        result.update({"stem": core[:-1], "suffix": "s", "pos_guess": "NOUN",␣
↪"features": {"number": "PLUR"}})
        return result

    # default
    result["stem"] = core
    return result
```

1.1.4 Try it: Analyze a few words

Run the next cell to see how the toy analyzer segments and tags common forms.

[2]:
```
examples = [
    "running", "hoped", "tries", "boxes", "children", "women",
    "cats", "better", "nicest", "quickly", "disagree", "went", "Saw"
]

for w in examples:
    print(analyze(w))
```

```
{'token': 'running', 'prefix': '', 'stem': 'run', 'suffix': 'ing', 'pos_guess':
'VERB', 'features': {'aspect': 'PROG'}}
{'token': 'hoped', 'prefix': '', 'stem': 'hope', 'suffix': 'ed', 'pos_guess':
'VERB', 'features': {'tense': 'PAST/PART'}}
{'token': 'tries', 'prefix': '', 'stem': 'try', 'suffix': 'ies', 'pos_guess':
'VERB', 'features': {'tense': 'PRES.3SG'}}
{'token': 'boxes', 'prefix': '', 'stem': 'box', 'suffix': 'es', 'pos_guess':
'VERB', 'features': {'tense': 'PRES.3SG'}}
{'token': 'children', 'prefix': '', 'stem': 'child', 'suffix': '', 'pos_guess':
'NOUN', 'features': {'number': 'PLUR'}}
{'token': 'women', 'prefix': '', 'stem': 'woman', 'suffix': '', 'pos_guess':
'NOUN', 'features': {'number': 'PLUR'}}
```

```
{'token': 'cats', 'prefix': '', 'stem': 'cat', 'suffix': 's', 'pos_guess':
'VERB', 'features': {'tense': 'PRES.3SG'}}
{'token': 'better', 'prefix': '', 'stem': 'good', 'suffix': '', 'pos_guess':
'ADJ', 'features': {'degree': 'COMP'}}
{'token': 'nicest', 'prefix': '', 'stem': 'nice', 'suffix': 'est', 'pos_guess':
'ADJ', 'features': {'degree': 'SUPER'}}
{'token': 'quickly', 'prefix': '', 'stem': 'quick', 'suffix': 'ly', 'pos_guess':
'ADV', 'features': {}}
{'token': 'disagree', 'prefix': 'dis', 'stem': 'agree', 'suffix': '',
'pos_guess': 'UNK', 'features': {}}
{'token': 'went', 'prefix': '', 'stem': 'go', 'suffix': '', 'pos_guess': 'VERB',
'features': {'tense': 'PAST'}}
{'token': 'Saw', 'prefix': '', 'stem': 'see', 'suffix': '', 'pos_guess': 'VERB',
'features': {'tense': 'PAST'}}
```

1.1.5 (Optional) Display results as a table

[5]:
```
import pandas as pd
examples = [""
    "running", "hoped", "tries", "boxes", "children", "women",
    "cats", "better", "nicest", "quickly", "disagree", "went","Budditha", "Saw"
]

rows = []
for w in examples:
    r = analyze(w)
    rows.append({
        "token": r["token"],
        "prefix": r["prefix"],
        "stem": r["stem"],
        "suffix": r["suffix"],
        "pos_guess": r["pos_guess"],
        "features": r["features"],
    })

df = pd.DataFrame(rows)
df
```

[5]:

	token	prefix	stem	suffix	pos_guess	features
0	running		run	ing	VERB	{'aspect': 'PROG'}
1	hoped		hope	ed	VERB	{'tense': 'PAST/PART'}
2	tries		try	ies	VERB	{'tense': 'PRES.3SG'}
3	boxes		box	es	VERB	{'tense': 'PRES.3SG'}
4	children		child		NOUN	{'number': 'PLUR'}
5	women		woman		NOUN	{'number': 'PLUR'}
6	cats		cat	s	VERB	{'tense': 'PRES.3SG'}
7	better		good		ADJ	{'degree': 'COMP'}

```
8     nicest             nice    est       ADJ     {'degree': 'SUPER'}
9    quickly            quick     ly       ADV                     {}
10  disagree    dis     agree              UNK                     {}
11      went               go             VERB       {'tense': 'PAST'}
12  Budditha         budditha              UNK                     {}
13       Saw              see             VERB       {'tense': 'PAST'}
```

1.1.6 ML Example

Machine learning example of English morphological analysis in a Jupyter-friendly way.

train a tiny character-level neural model (seq2seq with LSTM) to learn mappings like:

"running" → "run"

"hoped" → "hope"

"tries" → "try"

"cats" → "cat"

[7]:
```python
# pip install tensorflow scikit-learn

import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, TimeDistributed
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split

# -------------------------
# 1. Toy dataset (inflected → lemma)
# -------------------------
pairs = [
    ("running", "run"),
    ("hoped", "hope"),
    ("tries", "try"),
    ("cats", "cat"),
    ("dogs", "dog"),
    ("studies", "study"),
    ("boxes", "box"),
    ("better", "good"),
    ("went", "go"),
]

# collect all characters
chars = sorted(set("".join(w for pair in pairs for w in pair)))
char2idx = {c: i+1 for i, c in enumerate(chars)}  # 0 reserved for padding
idx2char = {i: c for c, i in char2idx.items()}

max_len = max(max(len(w1), len(w2)) for w1, w2 in pairs)
```

```
def encode_word(word):
    return [char2idx[c] for c in word]

X = [encode_word(w1) for w1, _ in pairs]
y = [encode_word(w2) for _, w2 in pairs]

X = pad_sequences(X, maxlen=max_len, padding="post")
y = pad_sequences(y, maxlen=max_len, padding="post")

# one-hot encode output
num_classes = len(char2idx) + 1
y_onehot = np.zeros((len(y), max_len, num_classes))
for i, seq in enumerate(y):
    for t, idx in enumerate(seq):
        if idx > 0:
            y_onehot[i, t, idx] = 1.0

# -------------------------
# 2. Build seq2seq-like model
# -------------------------
model = Sequential([
    Embedding(input_dim=num_classes, output_dim=32, input_length=max_len,␣
 ↪mask_zero=True),
    LSTM(64, return_sequences=True),
    TimeDistributed(Dense(num_classes, activation="softmax"))
])

model.compile(loss="categorical_crossentropy", optimizer="adam",␣
 ↪metrics=["accuracy"])
model.summary()

# -------------------------
# 3. Train
# -------------------------
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2,␣
 ↪random_state=42)
model.fit(X_train, y_train, batch_size=4, epochs=50, validation_data=(X_test,␣
 ↪y_test), verbose=0)

# -------------------------
# 4. Helpers for prediction
# -------------------------
def decode(seq):
    return "".join(idx2char.get(np.argmax(vec), "") for vec in seq if np.
 ↪argmax(vec) > 0)
```

```python
def predict(word):
    seq = pad_sequences([encode_word(word)], maxlen=max_len, padding="post")
    pred = model.predict(seq, verbose=0)
    return decode(pred[0])

# -------------------------
# 5. Test
# -------------------------
test_words = ["running", "cats", "studies", "hoped", "went", "dogs"]
for w in test_words:
    print(f"{w:10s} → {predict(w)}")
```

```
2025-09-12 05:16:16.780149: I tensorflow/core/util/port.cc:153] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-09-12 05:16:16.789110: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1757654176.798648 1654988 cuda_dnn.cc:8579] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1757654176.801521 1654988 cuda_blas.cc:1407] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered
W0000 00:00:1757654176.808808 1654988 computation_placer.cc:177] computation
placer already registered. Please check linkage and avoid linking the same
target more than once.
W0000 00:00:1757654176.808816 1654988 computation_placer.cc:177] computation
placer already registered. Please check linkage and avoid linking the same
target more than once.
W0000 00:00:1757654176.808817 1654988 computation_placer.cc:177] computation
placer already registered. Please check linkage and avoid linking the same
target more than once.
W0000 00:00:1757654176.808818 1654988 computation_placer.cc:177] computation
placer already registered. Please check linkage and avoid linking the same
target more than once.
2025-09-12 05:16:16.811382: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations,
rebuild TensorFlow with the appropriate compiler flags.
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97:
UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
```

```
2025-09-12 05:16:18.031083: E
external/local_xla/xla/stream_executor/cuda/cuda_platform.cc:51] failed call to
cuInit: INTERNAL: CUDA error: Failed call to cuInit: UNKNOWN ERROR (303)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
time_distributed (TimeDistributed)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
2025-09-12 05:16:19.262388: E tensorflow/core/util/util.cc:131] oneDNN supports
DT_BOOL only on platforms with AVX-512. Falling back to the default Eigen-based
implementation if present.
running    → ruuyyyy
cats       → ttttttt
studies    → ttudyyy
hoped      → toooooo
went       → ooooooo
dogs       → toogggg
```

1.1.7 Finite-State Transducer (FST) for English Morphological Analysis

Finite-State Transducer (FST) for English Morphological Analysis using Pynini (Python bindings for OpenFST). It supports both generation (lexical → surface) and analysis (surface → lexical) for a handful of common patterns:

- Noun plural: cat+N+PL → cats, box+N+PL → boxes, study+N+PL → studies
- Verb 3rd-sg: play+V+3SG → plays, try+V+3SG → tries, fix+V+3SG → fixes
- Verb past: play+V+PAST → played, try+V+PAST → tried, hope+V+PAST → hoped, and irregular go+V+PAST → went

- Verb progressive: play+V+PROG → playing, hope+V+PROG → hoping, study+V+PROG → studying (drop-e before -ing)

[10]:
```
# English Morphology with Finite-State Transducers (Pynini)
# Features: N+PL, V+3SG, V+PAST, V+PROG (+ drop-e, y→ies/ied, sibilant+es)
# Includes irregular example: go+V+PAST -> went

import pynini
from pynini.lib import rewrite

# ----- Alphabet (sigma) -----
letters = list("abcdefghijklmnopqrstuvwxyz")

# Build a permissive sigma (alphabet) without using acceptor()
SIGMA = pynini.union(*letters, "^", "P","L","3","S","A","T","R","O","G").
 ↪closure().optimize()

# Helpful sets
V = pynini.union("a", "e", "i", "o", "u")
C = pynini.union(*[c for c in letters if c not in "aeiou"])
SIBILANT = pynini.union("s", "x", "z", "ch", "sh")  # for +es rules

# ----- 1) Morphotactics: lexical → stem + feature marker -----
# Map "lemma+POS+FEAT" → "lemma^FEAT" (or directly to an irregular surface).
noun_pl = [
    ("cat+N+PL",   "cat^PL"),
    ("box+N+PL",   "box^PL"),
    ("study+N+PL", "study^PL"),
]

verb_3sg = [
    ("play+V+3SG", "play^3S"),
    ("try+V+3SG",  "try^3S"),
    ("fix+V+3SG",  "fix^3S"),
]

verb_past = [
    ("play+V+PAST", "play^PAST"),
    ("try+V+PAST",  "try^PAST"),
    ("hope+V+PAST", "hope^PAST"),
    ("go+V+PAST",   "went"),        # irregular path
]

verb_prog = [
    ("play+V+PROG",  "play^PROG"),
    ("hope+V+PROG",  "hope^PROG"),
    ("study+V+PROG", "study^PROG"),
```

```
]

LEX2MARK = pynini.string_map(noun_pl + verb_3sg + verb_past + verb_prog).
 ↪optimize()

# ----- 2) Orthographic rules: markers → surface -----
rules = []

# Noun plural
rules.append(pynini.cdrewrite(pynini.cross("y^PL", "ies"), "", "", SIGMA))      ␣
 ↪      # study → studies
rules.append(pynini.cdrewrite(pynini.cross("^PL",  "es"),  SIBILANT, "",␣
 ↪SIGMA))      # box → boxes
rules.append(pynini.cdrewrite(pynini.cross("^PL",  "s"),   "", "", SIGMA))      ␣
 ↪      # cat → cats

# Verb 3SG
rules.append(pynini.cdrewrite(pynini.cross("y^3S", "ies"), "", "", SIGMA))      ␣
 ↪      # try → tries
rules.append(pynini.cdrewrite(pynini.cross("^3S",  "es"),  SIBILANT, "",␣
 ↪SIGMA))      # fix → fixes
rules.append(pynini.cdrewrite(pynini.cross("^3S",  "s"),   "", "", SIGMA))      ␣
 ↪      # play → plays

# Verb PAST
rules.append(pynini.cdrewrite(pynini.cross("y^PAST", "ied"), "", "", SIGMA))    ␣
 ↪      # try → tried
rules.append(pynini.cdrewrite(pynini.cross("^PAST",  "ed"),  "", "", SIGMA))    ␣
 ↪      # play → played

# Verb PROG (-ing)
rules.append(pynini.cdrewrite(pynini.cross("e^PROG", "^PROG"), "", "", SIGMA)) ␣
 ↪      # hope → hop^PROG
rules.append(pynini.cdrewrite(pynini.cross("^PROG",  "ing"),  "", "", SIGMA))   ␣
 ↪      # ^PROG → ing

# Compose specific → general (order matters)
ORTHO = SIGMA
for r in rules:
    ORTHO = (ORTHO @ r)
ORTHO = ORTHO.optimize()

# ----- 3) Build Generation & Analysis FSTs -----
GEN = (LEX2MARK @ ORTHO).optimize()    # lexical → surface
ANALYZE = GEN.invert().optimize()      # surface → lexical
```

```python
# ----- 4) Helpers -----
def gen(lexical):
    try:
        return rewrite.top_rewrite(lexical, GEN)
    except rewrite.Error:
        return "  (no generation)"

def ana(surface):
    try:
        return rewrite.top_rewrite(surface, ANALYZE)
    except rewrite.Error:
        return "  (no analysis)"

# ----- 5) Demo -----
tests_gen = [
    "cat+N+PL", "box+N+PL", "study+N+PL",
    "play+V+3SG", "try+V+3SG", "fix+V+3SG",
    "play+V+PAST", "try+V+PAST", "hope+V+PAST", "go+V+PAST",
    "play+V+PROG", "hope+V+PROG", "study+V+PROG",
]

tests_ana = [
    "cats","boxes","studies",
    "plays","tries","fixes",
    "played","tried","hoped","went",
    "playing","hoping","studying",
]

print("=== Generation (lex → surf) ===")
for t in tests_gen:
    print(f"{t:18s} -> {gen(t)}")

print("\n=== Analysis (surf → lex) ===")
for t in tests_ana:
    print(f"{t:10s} -> {ana(t)}")
```

```
=== Generation (lex → surf) ===
cat+N+PL           ->   (no generation)
box+N+PL           ->   (no generation)
study+N+PL         ->   (no generation)
play+V+3SG         ->   (no generation)
try+V+3SG          ->   (no generation)
fix+V+3SG          ->   (no generation)
play+V+PAST        ->   (no generation)
try+V+PAST         ->   (no generation)
hope+V+PAST        ->   (no generation)
go+V+PAST          ->   (no generation)
play+V+PROG        ->   (no generation)
```

```
hope+V+PROG        ->   (no generation)
study+V+PROG       ->   (no generation)

=== Analysis (surf → lex) ===
cats       -> cat+N+PL
boxes      -> box+N+PL
studies    -> study+N+PL
plays      ->   (no analysis)
tries      -> try+V+3SG
fixes      -> fix+V+3SG
played     ->   (no analysis)
tried      -> try+V+PAST
hoped      ->   (no analysis)
went       -> go+V+PAST
playing    -> play+V+PROG
hoping     -> hope+V+PROG
studying   -> study+V+PROG
```

[ ]:
```
### tiny, rule-based Sinhala morphological analyzer
```

[11]:
```
# -*- coding: utf-8 -*-

from dataclasses import dataclass, asdict
from typing import Dict, List, Tuple

@dataclass
class Analysis:
    token: str
    lemma: str
    pos: str
    features: Dict[str, str]
    segments: Dict[str, List[str]]

# --- Suffix lists (very small pedagogical subset) ---

# Noun plural (common patterns)
PLURALS: List[Tuple[str, str]] = [
    (" ", "PL.INAN"),   #   ,
    (" ",  "PL.HUM"),    #    → (varies), but ' / ' also occur; we keep   for␣
 ↪teaching
]

# Case/particle markers (colloquial + formal subset)
CASE_SUFFIXES: List[Tuple[str, str]] = [
    (" ", "GEN"),   #
    (" ",  "DAT"),   #
    (" ",  "ACC/DEF"),#     (object/definite marker)
```

```
    (" ", "INS/ABL"), #    /
    (" ", "LOC"),   #      (more formal/written)
    (" ", "LOC"),   #
]

# Definite / honorific-like endings (very simplified)
DET_SUFFIXES: List[Tuple[str, str]] = [
    (" ", "DEF.MASC"),   #
    ("  ", "HON"),      #     ,
]

# Verb endings (very simplified)
VERB_SUFFIXES: List[Tuple[str, str]] = [
    ("  ", "V.PRES"),  #    ,     → stem   colloquial root
    (" ", "V.GER"),  #   ,
    (" ",   "V.PRES.FML"), #    (formal/literary)
]

# Order matters: strip longer suffixes first
PLURALS.sort(key=lambda x: len(x[0]), reverse=True)
CASE_SUFFIXES.sort(key=lambda x: len(x[0]), reverse=True)
DET_SUFFIXES.sort(key=lambda x: len(x[0]), reverse=True)
VERB_SUFFIXES.sort(key=lambda x: len(x[0]), reverse=True)

def strip_one(word: str, table: List[Tuple[str,str]]):
    for suf, tag in table:
        if word.endswith(suf) and len(word) > len(suf):
            return word[:-len(suf)], (suf, tag)
    return word, None

def analyze_sinhala_token(tok: str) -> Analysis:
    w = tok.strip()
    seg_prefixes: List[str] = []   # (not used in this toy)
    seg_suffixes: List[str] = []
    feats: Dict[str,str] = {}

    # 1) Try verb analysis first (common in texts)
    for suf, tag in VERB_SUFFIXES:
        if w.endswith(suf) and len(w) > len(suf):
            stem = w[:-len(suf)]
            seg_suffixes.append(suf)
            feats["verb_form"] = tag
            # very rough lemma heuristic:
            # if V.PRES (  ), we use the stem as lemma (e.g., "   " → " ")
            lemma = stem
            return Analysis(token=tok, lemma=lemma, pos="VERB",
```

```
                                features=feats, segments={"prefixes": seg_prefixes,␣
 ↪"stem":[lemma], "suffixes": seg_suffixes})

    # 2) Noun: strip plural → case/particle → definite/honorific
    pos_guess = "NOUN"
    lemma = w

    # plural
    lemma, pl = strip_one(lemma, PLURALS)
    if pl:
        seg_suffixes.append(pl[0])
        feats["number"] = pl[1]

    # case / object / loc
    lemma, kase = strip_one(lemma, CASE_SUFFIXES)
    if kase:
        seg_suffixes.append(kase[0])
        feats["case"] = kase[1]

    # definiteness/honorific
    lemma, det = strip_one(lemma, DET_SUFFIXES)
    if det:
        seg_suffixes.append(det[0])
        feats["def/hon"] = det[1]

    return Analysis(token=tok, lemma=lemma, pos=pos_guess,
                    features=feats, segments={"prefixes": seg_prefixes, "stem":
 ↪[lemma], "suffixes": seg_suffixes})

# --- Demo ---
examples = [
    "     ",  # teacher+DEF.MASC + GEN
    "   ",      # books (inanimate plural)
    "  ",        # son + DAT
    "   ",       # say + PRES (colloquial)
    "  ",       # saying (gerund)
    "   ",       # in/at Colombo (LOC)
    "    ",    # friends' (not fully covered; shows limitation)
]

for t in examples:
    a = analyze_sinhala_token(t)
    print(asdict(a))
```

```
{'token': '     ', 'lemma': '  ', 'pos': 'NOUN', 'features': {'case':
'GEN', 'def/hon': 'DEF.MASC'}, 'segments': {'prefixes': [], 'stem': ['  '],
'suffixes': [' ', ' ']}}
```

```
{'token': '   ', 'lemma': '  ', 'pos': 'NOUN', 'features': {'number':
'PL.INAN'}, 'segments': {'prefixes': [], 'stem': ['  '], 'suffixes': [' ']}}
{'token': '  ', 'lemma': '  ', 'pos': 'NOUN', 'features': {'case': 'DAT'},
'segments': {'prefixes': [], 'stem': ['  '], 'suffixes': [' ']}}
{'token': '   ', 'lemma': ' ', 'pos': 'VERB', 'features': {'verb_form':
'V.PRES'}, 'segments': {'prefixes': [], 'stem': [' '], 'suffixes': ['  ']}}
{'token': '  ', 'lemma': ' ', 'pos': 'VERB', 'features': {'verb_form':
'V.GER'}, 'segments': {'prefixes': [], 'stem': [' '], 'suffixes': [' ']}}
{'token': '   ', 'lemma': '  ', 'pos': 'NOUN', 'features': {'case': 'LOC'},
'segments': {'prefixes': [], 'stem': ['  '], 'suffixes': [' ']}}
{'token': '    ', 'lemma': '  ', 'pos': 'NOUN', 'features': {'case':
'GEN'}, 'segments': {'prefixes': [], 'stem': ['  '], 'suffixes': [' ']}}
```

[ ]: