Functional Programming in Python for Linguists

1. Functional Programming

What is Functional Programming?

- Functional programming (FP) is a **programming paradigm** that focuses on **pure functions, immutability, and avoiding side effects**. Unlike imperative programming, where you define a sequence of commands that modify the program's state, functional programming treats computation as the evaluation of mathematical functions.

Contd.

Key Principles of Functional Programming

1. First-Class Functions

1. Functions can be assigned to variables, passed as arguments, and returned from other functions.
2. This makes code more modular and reusable.

2. Immutability

1. Once a variable is assigned, it **cannot** be changed.
2. Instead of modifying existing data, FP creates new data structures.

Contd.

3. **Pure Functions**

- A function's output depends only on its input and has **no side effects** (i.e., it does not modify external variables or data).
- This makes code easier to test and debug.

4. **Higher-Order Functions**

- Functions that take other functions as arguments or return functions.
- Common examples: *map(), filter(),* and *reduce().*

Contd.

5. **Recursion Instead of Loops**

- Instead of using *for* or *while* loops, FP relies on **recursive function calls.**

6. **Lazy Evaluation**

- Expressions are not evaluated until their values are needed.
- This improves performance and optimizes memory usage.

Contd.

Why is Functional Programming Useful in Linguistics?

- Linguistics involves processing large amounts of text, such as corpora, dictionaries, and phonological patterns.
- Functional programming is particularly useful because it:

1. **Handles Large Datasets Efficiently**

- Linguistic data (like corpora or speech datasets) can be processed in parallel using functional constructs like *map()*.

2. **Encourages Modular & Reusable Code**

- Functional programming allows you to write small, reusable functions.
- Example: A function that converts text to lowercase can be reused in multiple linguistic analyses.

Contd.

3. **Avoids Unintended Side Effects**

- Linguistic processing often involves **multiple transformations** on text (e.g., tokenization, stemming, lemmatization).
- FP ensures **data remains unchanged**, preventing accidental corruption.

4. **Supports Parallel Processing**

- Corpus linguistics and sentiment analysis require processing **millions of words.**
- Functional programming supports **parallel execution** (e.g., *multiprocessing* in Python).

Contd.

Example: Imperative vs. Functional Style in Text Processing

Consider a task where we need to **filter out stopwords** from a sentence.

- **Imperative Style (Traditional Approach)**

```python
stopwords = {"a", "the", "in", "on", "and", "is"}
sentence = "This is a simple example"

words = sentence.split()
filtered_words = []

for word in words:
    if word not in stopwords:
        filtered_words.append(word)

print(filtered_words)
```

```
['This', 'simple', 'example']
```

Contd.

Problems with Imperative Style

- Uses a loop and mutable lists (*filtered_words*).
- Code is longer and harder to debug.

Contd.

- **Functional Style** (Using *filter()* and *lambda*)

```
stopwords = {"a", "the", "in", "on", "and", "is"}
sentence = "This is a simple example"

filtered_words = list(filter(lambda w: w not in stopwords, sentence.split()))

print(filtered_words)
```

- Output: `['This', 'simple', 'example']`

➢ **Why is this Better?**

  - **More concise:** Eliminates explicit loops.
  - **Immutable:** No unnecessary modifications.
  - **More readable & reusable.**

2. Variables, State, and Mutability

Understanding Immutability

- In functional programming, variables don’t change once assigned.
- Helps prevent bugs caused by unexpected state changes.

Contd.

- **Example**:

```python
x = "hello"
y = x.upper()
print(y)   # HELLO
print(x)   # hello (unchanged)
```

Why Avoid State Changes?

- Reduces side effects.
- Makes debugging easier.

3. Functions as First-Class Objects

- **What Does "First-Class Citizen" Mean?**
  - Functions can be stored in variables, passed as arguments, and returned from other functions.
- **Example:**

```python
def greet(name):
    return "Hello, " + name

welcome = greet  # Assign function to a variable
print(welcome("Alice"))  # Output: Hello, Alice
```

- **Linguistics Application:**
  - Defining functions for text transformations.

4. Lambda Functions

What is a Lambda Function?

- A small anonymous function using the *lambda* keyword. Unlike regular functions created with def, lambda functions do not have a name and are typically used for short, simple operations.

➢ **Syntax**: lambda arguments: expression

  - lambda is the keyword.
  - arguments are the input parameters (like a normal function).
  - expression is the single operation the function performs (must return a value).

- **Key Difference from def Functions**:
  - Lambda functions are one-liners (cannot contain multiple statements).
  - Implicit return (no need to use return keyword).

Contd.

- **Example: Converting Words to Uppercase**
- Using a regular function (def):

```python
def uppercase(word):
    return word.upper()

print(uppercase("phoneme"))  # Output: PHONEME
```

- Using a lambda function:

```python
uppercase = lambda word: word.upper()
print(uppercase("phoneme"))  # Output: PHONEME
```

- **Why Use Lambda Functions?**
  - Shortens simple functions.
  - Useful in functions like *map()* and *filter().*

5. Overt Recursion

What is Recursion?

- A function that **calls itself until a base condition is met**.
- A recursive function **breaks down a complex problem into smaller, simpler subproblems**.
- Every recursive function must have a **base case** (a stopping condition) and **a recursive step** (where it calls itself).

Contd.

- **Example: Counting Syllables Recursively**

```python
def count_vowels(word, count=0):
    if word == "":
        return count
    return count_vowels(word[1:], count + (1 if word[0] in "aeiou" else 0))

print(count_vowels("phonology"))  # Output: 3
```

➢Why is Recursion Useful in Linguistics?

- Parsing Nested Structures (e.g., Sentence Trees)
- Morphological Analysis (Breaking Words into Morphemes)
- Phonological Rule Application

6. Comprehensions in Python

- Python comprehensions allow for **concise and efficient** creation of **lists, sets, and dictionaries** in a single line.
- They are faster and more readable than traditional loops.

Contd.

- **Types of Comprehensions**

Comprehension Type	Syntax	Example
List Comprehension	[expression for item in iterable if condition]	[w for w in words if "ing" in w]
Set Comprehension	{expression for item in iterable if condition}	{len(w) for w in words}
Dictionary Comprehension	{key: value for item in iterable if condition}	{w: len(w) for w in words}

Contd.

- **Example: Finding Words with 'ing' Using List Comprehension**

```python
words = ["running", "jump", "singing", "play"]
ing_words = [w for w in words if "ing" in w]
print(ing_words)  # Output: ['running', 'singing']
```

- **Equivalent Traditional Loop**

```python
words = ["running", "jump", "singing", "play"]
ing_words = []
for w in words:
    if "ing" in w:
        ing_words.append(w)
print(ing_words)  # Output: ['running', 'singing']
```

Why use List Comprehension?

- **More concise** than loops.
- **Faster** than appending to lists manually.

Contd.

- Comprehensions use **optimized internal loops**, making them **faster** than traditional loops.

Performance Comparison

```
import time

words = ["morphology", "syntax", "phonetics"] * 1000  # Large dataset

# Using a Loop
start = time.time()
word_lengths = {}
for w in words:
    word_lengths[w] = len(w)
end = time.time()
print(f"Loop Time: {end - start:.6f} sec")

# Using Dictionary Comprehension
start = time.time()
word_lengths = {w: len(w) for w in words}
end = time.time()
print(f"Comprehension Time: {end - start:.6f} sec")
```

Contd.

- **Expected Output** (Varies by System):

```
Loop Time: 0.000420 sec
Comprehension Time: 0.000308 sec
```

📌 **Use comprehensions whenever possible** for simple transformations—they make code **faster and cleaner!**

Contd.

- **Nested Comprehensions:**

Comprehensions can also **replace nested loops** for more complex operations.

- **Example: Creating Word Bigrams Using Nested List Comprehension**

```python
words = ["syntax", "morphology"]
bigrams = [(w[i], w[i+1]) for w in words for i in range(len(w) - 1)]
print(bigrams)
# Output: [('s', 'y'), ('y', 'n'), ('n', 't'), ('t', 'a'), ...]
```

Why use nested comprehensions?

- **Avoids deeply nested loops.**
- **More efficient** in handling multiple iterations.

Contd.

- **Why Use Comprehensions?**

Feature	Traditional Loops	Comprehensions
Code Length	Longer	Shorter
Readability	More verbose	More Concise
Performance	Slower	Faster
Best For	Complex Logic	Simple Transformations

💡 **Rule of Thumb:**

- ✅ Use **comprehensions** for simple list, set, or dictionary transformations.
- ❌ Use **loops** when code requires multiple statements or complex logic.

7. Vectorized Computation in Python

- Vectorized computation is a technique where **operations are applied to entire arrays (vectors) at once, instead of using loops**.
- **Key Features:**
  - ✓Eliminates explicit loops, making code more **concise and readable**.
  - ✓Uses **optimized low-level implementations** for speed (e.g., NumPy in Python).
  - ✓Ideal for processing **large linguistic datasets** efficiently.

7.1. Using NumPy for Linguistics

Python's built-in lists are **not optimized for numerical operations**. Instead, **NumPy arrays** allow efficient vectorized computations.

- **Example: Token Frequency in a Corpus**

```python
import numpy as np

# Word frequencies from a corpus
freqs = np.array([10, 50, 30, 60])

# Normalize frequencies (convert to probabilities)
norm_freqs = freqs / np.sum(freqs)

print(norm_freqs)
```

- **How does it work?**
  - np.array([10, 50, 30, 60]) creates a NumPy array.
  - np.sum(freqs) calculates the total frequency.
  - freqs / np.sum(freqs) divides each element by the total → producing normalized frequencies.

No loops required! NumPy performs element-wise division faster than traditional loops.

- **Output**:

```
[0.06666667 0.33333333 0.2        0.4       ]
```

Contd.

- **Why is Vectorized Computation Useful?**

1. Speeds Up Large-Scale Linguistic Computations

- Processing **millions of words** in a **text corpus** is computationally expensive. NumPy **optimizes operations using C and Fortran**, making it significantly faster.

2. More Efficient Than Traditional Loops

- **Loop-Based Approach (Slow)**

```python
freqs = [10, 50, 30, 60]
total = sum(freqs)
norm_freqs = [f / total for f in freqs]  # List comprehension
print(norm_freqs)
```

- 🚨 **Problem:** Python lists require **explicit looping**, which is **slower** for large datasets.

Contd.

- **NumPy Approach (Fast & Efficient)**

```python
import numpy as np
freqs = np.array([10, 50, 30, 60])
norm_freqs = freqs / np.sum(freqs)   # Vectorized computation
print(norm_freqs)
```

✅ **No loops** → Operations happen at the **hardware level** (C/Fortran), making it **much faster**.

Contd.

Example: Text Processing with NumPy

Let's say we have a list of **word lengths** and we want to find the **mean length**.

- **Loop-Based Approach:**

```
word_lengths = [4, 7, 5, 6, 8]
mean_length = sum(word_lengths) / len(word_lengths)
print(mean_length)  # Output: 6.0
```

- **NumPy Approach (Faster & Cleaner):**

```
import numpy as np
word_lengths = np.array([4, 7, 5, 6, 8])
mean_length = np.mean(word_lengths)  # Vectorized mean calculation
print(mean_length)  # Output: 6.0
```

✅ NumPy computes directly without a loop!

Contd.

- **Real-World Linguistic Applications of Vectorized Computation**

Application	How Vectorization Helps
Token Frequency Analysis	Normalizing word counts without loops
Phonetic Feature Processing	Vectorized transformations of speech signals
Corpus Statistics	Fast computation of mean word length, type-token ratio, etc.
TF-IDF Computation	Computing term frequency-inverse document frequency efficiently

Contd.

- **Why Use Vectorized Computation?**

Feature	Loops (Traditional Approach)	Vectorized (NumPy)
Speed	Slower (executes element by element)	Faster (executes at C-level)
Code Simplicity	Requires multiple lines of code	Achieves the same in one line
Memory Efficiency	Stores intermediate results	Optimized for memory usage
Performance	Iterates in Python (slower)	Uses optimized C/Fortran operations
Scalability	Becomes slow with large datasets	Handles large corpora efficiently

8. Iterables, Iterators, and Generators in Python

What is an Iterable?

- An **iterable** is any object in Python that can be **looped over** (e.g., lists, tuples, strings, dictionaries, etc.).
- It contains elements that can be accessed **one by one** but does not keep track of where it is in the iteration.
- **Example: Lists (An Iterable)**

```python
words = ["syntax", "morphology", "phonetics"]
for w in words:
    print(w)
```

✔️ Works with loops (e.g., for loops)

❌ Does not remember position once exhausted

Contd.

What is an Iterator?

- An iterator is an object that **remembers its position** in the sequence while iterating.
  - It must implement *__iter__()* and *__next__()* methods.
  - Once an iterator is exhausted, it **cannot be reset** (unlike iterables).
- **Example: Creating an Iterator from an Iterable**

```python
words = ["syntax", "morphology", "phonetics"]
iterator = iter(words)  # Convert list into an iterator

print(next(iterator))  # Output: syntax
print(next(iterator))  # Output: morphology
print(next(iterator))  # Output: phonetics
```

❌ Calling next(iterator) again will raise an error (StopIteration).

Contd.

What is a Generator?

- A **generator** is a special type of iterator that **produces values lazily**, meaning it **yields one value at a time** instead of storing everything in memory.

✔ **Uses yield instead of return**
✔ **Automatically remembers its state**
✔ **More memory-efficient than lists**

Contd.

- **Example: Generator Function for Streaming Words**

```python
def word_stream():
    words = ["syntax", "morphology", "phonetics"]
    for w in words:
        yield w  # Produces one word at a time

stream = word_stream()  # Create generator object

print(next(stream))  # Output: syntax
print(next(stream))  # Output: morphology
print(next(stream))  # Output: phonetics
```

➢ **How Does It Work?**

Calling word_stream() **does not execute the function immediately** → It creates a **generator object**.

The first next(stream) **executes until it reaches yield**, returning "syntax".

The second next(stream) **resumes execution** where it left off, returning "morphology".

Contd.

- **Why Use Generators?**

✅ **Memory-Efficient for Large Text Processing**

- Instead of **loading an entire corpus into memory**, a generator **fetches one word at a time**.

✅ **Faster Performance**

- No need to **precompute and store** all values → **Improves speed** for big data.

✅ **Can be used in Streaming Data Processing**

- Perfect for handling **large files, corpora, and databases** without **high RAM usage**.

Contd.

- **Example: Reading a Large Text File Line by Line (Efficiently)**

```python
def read_large_file(filename):
    with open(filename, "r", encoding="utf-8") as file:
        for line in file:
            yield line.strip()  # Yield one line at a time

lines = read_large_file("corpus.txt")

print(next(lines))  # Prints the first line of the file
print(next(lines))  # Prints the second line
```

✅ Does not load the entire file into memory!

Contd.

- **Why Use Iterators & Generators?**

Feature	Iterable (List, String)	Iterator (iter())	Generator (yield)
Loop Over Data	✅ Yes	✅ Yes	✅ Yes
Remembers Position	❌ No	✅ Yes	✅ Yes
Exhaustible?	❌ No	✅ Yes	✅ Yes
Memory Efficiency	❌ Loads all data	❌ Stores all data	✅ Generates on demand
Best For	Small data sets	Keeping track of iteration	Large-scale text processing

Contd.

Key Takeaways

- **Use Iterables** (list, tuple, string) when working with **small datasets**.
- **Use Iterators** when you need to **manually control iteration**.
- **Use Generators** when working with **large text files, corpora, or real-time data streaming**.

9. Parallel Programming in Python

- Parallel processing **speeds up tasks** by executing multiple operations **simultaneously** instead of one after another (sequential processing).
- **Key Benefits:**
  - **Faster execution** → Speeds up **corpus processing, NLP tasks, and text analysis**.
  - **Utilizes multiple CPU cores** → More efficient for **large datasets**.
  - **Ideal for repetitive linguistic computations** (e.g., phoneme classification, token frequency calculations).

Contd.

Using multiprocessing to Process Words in Parallel

- Python's multiprocessing module **allows multiple CPU cores to process data concurrently**. Below, we convert words **to uppercase** in **parallel**, using two CPU cores.

```python
from multiprocessing import Pool

def process_word(word):
    return word.upper()   # Convert to uppercase

words = ["phoneme", "syllable", "morpheme"]

# Create a pool of 2 worker processes
with Pool(2) as p:
    results = p.map(process_word, words)   # Process words in parallel

print(results)   # Output: ['PHONEME', 'SYLLABLE', 'MORPHEME']
```

- **How Does It Work?**
  Pool(2) → Creates a **pool of 2 worker processes**.
  map(process_word, words) → **Splits the list** across processes and applies process_word().
  Each process runs independently, utilizing multiple CPU cores.

Contd.

- **Why Use multiprocessing Instead of a Loop?**

Feature	Loop (Sequential)	Multiprocessing (Parallel)
Speed	Slower (one task at a time)	Faster (multiple tasks at once)
CPU Usage	Uses one core	Uses multiple cores
Efficiency	Good for small tasks	Best for large datasets
Use Case	Simple word processing	Large-scale corpus analysis

Contd.

Real-World Use Cases in Linguistics

- **Fast Corpus Processing (e.g., Tokenizing a Large Text Corpus)**

```python
from multiprocessing import Pool
import nltk

nltk.download("punkt")
from nltk.tokenize import word_tokenize

def tokenize_sentence(sentence):
    return word_tokenize(sentence)

sentences = [
    "Phonetics studies the sounds of speech.",
    "Morphology examines word structure.",
    "Syntax analyzes sentence structure."
]

# Process sentences in parallel
with Pool(3) as p:
    tokenized_sentences = p.map(tokenize_sentence, sentences)

print(tokenized_sentences)
```

✅ Speeds up tokenization for large corpora!

Contd.

Parallel Phoneme Classification

- Instead of looping over **millions of phonemes**, we can **process them in parallel**.

```python
from multiprocessing import Pool

phonemes = ["a", "e", "i", "o", "u"]

def classify_phoneme(phoneme):
    vowels = {"a", "e", "i", "o", "u"}
    return (phoneme, "vowel" if phoneme in vowels else "consonant")

with Pool(2) as p:
    classified_phonemes = p.map(classify_phoneme, phonemes)

print(classified_phonemes)
# Output: [('a', 'vowel'), ('e', 'vowel'), ('i', 'vowel'), ('o', 'vowel'), ('u', 'vowel')]
```

✅ Handles large phoneme datasets efficiently!

Contd.

When NOT to Use Parallel Processing

❌ If a task **doesn't require high computation**, parallelization may **add overhead** instead of speeding things up.

✅ Use parallel processing for **CPU-intensive tasks**, like **text analysis, speech recognition, and NLP**.

Contd.

- **Final Takeaways:**
  - Parallel processing **speeds up** large-scale linguistic tasks.
  - Use **multiprocessing.Pool** for CPU-bound tasks.
  - Ideal for **tokenization, phoneme classification**, and **corpus analysis**.

10. Making Nonsense Items Again

Nonsense items are **artificially generated words** that **follow linguistic rules** but do **not exist in the language**.

Why Create Nonsense Words?

- Used in **psycholinguistic experiments** to test **word recognition**.
- Helps in studying **phonotactic constraints** (rules about which sounds can appear together).
- Useful for **modeling language acquisition** (how speakers process new words).

Contd.

Using Functional Programming to Generate Fake Words

- We can use **functional programming** (lambda, random.choice()) to generate nonsense words.

```python
import random

# Define possible consonants and vowels
consonants = "ptkbdgmn"
vowels = "aeiou"

# Generate a nonsense word (CVC pattern)
generate_word = lambda: random.choice(consonants) + random.choice(vowels) + random.choice(consonants)

# Generate and print a random word
print(generate_word())  # Example Output: "tam"
```

How Does It Work?

- **random.choice(consonants)** picks a random consonant.
- **random.choice(vowels)** picks a random vowel.
- **random.choice(consonants)** picks another consonant → forming a **CVC (Consonant-Vowel-Consonant) structure**.

Contd.

Linguistics Application: Phonotactic Constraints

- Phonotactic constraints determine **which sound sequences are allowed** in a language.
- **Example: Generating Nonsense Words for English**
  - Some consonant clusters, like "bn" or "zr", are **not allowed** in English.
    We can filter **only valid English-like sequences**:

Contd.

```python
import random

# Define consonants and vowels with English phonotactic constraints
onset = "ptkbdgmn"  # Allowed initial consonants
vowel = "aeiou"
coda = "tksmn"  # Allowed final consonants

# Generate nonsense words following English-like patterns
generate_word = lambda: random.choice(onset) + random.choice(vowel) + random.choice(coda)

print(generate_word())  # Example Output: "tam"
```

✅ Ensures phonotactically valid words!

Contd.

Expanding to Complex Word Structures

- We can generate longer nonsense words (e.g., CCVCVCC patterns):

```python
import random

# Consonant clusters
clusters = ["bl", "tr", "st", "gr", "pl"]
vowels = "aeiou"

# Generate a nonsense word (CCVCVCC pattern)
generate_complex_word = lambda: random.choice(clusters) + random.choice(vowels) + random.choice(clusters)

print(generate_complex_word())  # Example Output: "plotr"
```

- **Why Is This Useful?**
  ✅ Simulates **natural language patterns**
  ✅ Used in **linguistic experiments**
  ✅ Can be **customized for different languages**

Contd.

Why Generate Nonsense Words?

Application	How Does It Help?
Psycholinguistics	Test word recognition & learning
Phonology	Study phonotactic rules
Speech Disorders	Assess pronunciation skills
Computational Linguistics	Generate test data for NLP models

11. Summary

Key Takeaways

- Functional programming avoids side effects and uses immutability.
- Recursion and comprehensions simplify code.
- Parallelism speeds up linguistic data processing.