Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. While Python is primarily an object-oriented language, it has robust support for functional programming constructs, allowing developers to write clean, concise, and expressive code that embraces functional principles.

Core Concepts of Functional Programming

While functional programming heavily relies on the concept of functions, the term "list programming" might sometimes feel more appropriate due to the emphasis on processing collections of data. Many functional programming constructs, such as map, filter, and reduce, are intrinsically tied to lists or other iterable structures. These operations often revolve around transforming, filtering, or aggregating sequences in a declarative manner, making the handling of lists central to the paradigm. This focus on working with lists highlights the strong connection between functional programming and data pipelines.

Here are the fundamental principles of functional programming.

- **Pure Functions**: A pure function always produces the same output for the same input and has no side effects (e.g., modifying variables outside its scope or interacting with I/O). This predictability makes programs easier to debug and reason about.
- **First-Class Functions**: In Python, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions. This enables higher-order functions.
- **Higher-Order Functions**: These are functions that take other functions as arguments or return them as results. Examples in Python include map, filter, and reduce.
- **Immutability**: Functional programming emphasizes the use of immutable data structures, avoiding changes to objects after they are created.
- **Recursion**: Instead of loops, functional programming often uses recursion to achieve iteration. However, Python's recursion depth limit can make this less practical in some cases.
- **Function Composition**: This involves combining simple functions to build more complex ones, allowing for reusable and modular code.

Iterables

In Python, an **iterable** is any object capable of returning its elements one at a time, allowing it to be looped over. It represents a collection of items, such as a list, tuple, string, or a custom object, that can produce its elements sequentially. Iterables are central to Python's data handling and form the foundation of loops, comprehensions, and many functional programming constructs.

Key Characteristics of an Iterable

1. **The __iter__ Method**: An iterable must implement the special method __iter__, which returns an iterator. This is the defining characteristic of an iterable.
2. **The __getitem__ Method**: Alternatively, objects that implement __getitem__ and support sequential access by index are also considered iterables.
3. **Not Exhausted**: Unlike iterators, iterables can be iterated over multiple times because they produce a new iterator each time __iter__ is called.

Examples of Iterables

- **Built-in Collections**: Lists, tuples, strings, dictionaries, and sets are all iterables.
- numbers = [1, 2, 3]
- for num in numbers:
-     print(num)
- **Custom Iterables**: You can define your own iterable by implementing the __iter__ method.

Iterable vs. Iterator

An **iterator** is a specific type of object associated with an iterable. While an iterable provides the structure (a collection of items), the iterator provides the mechanism to traverse that structure. When you call iter() on an iterable, it returns an iterator.

For example:

```
numbers = [1, 2, 3]

iterator = iter(numbers)  # Create an iterator from the list

print(next(iterator))     # Output: 1

print(next(iterator))     # Output: 2
```

Once an iterator is exhausted, it cannot be reused, whereas the original iterable can create a new iterator.

Custom Iterable Example

Here's how you can create your own iterable by defining the __iter__ and __next__ methods:

```
class MyIterable:

    def __init__(self, start, end):

        self.start = start

        self.end = end


    def __iter__(self):

        self.current = self.start

        return self
    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        value = self.current
        self.current += 1
        return value

# Using the custom iterable
for num in MyIterable(1, 5):
    print(num)
```

This custom class defines an iterable that generates numbers from start to end - 1.

Why Iterables Are Important

- **For Loops**: Iterables allow for loops to process items in a sequence.
- **Comprehensions**: List, set, and dictionary comprehensions work on iterables.
- **Functional Constructs**: Functions like map, filter, and zip operate on iterables.
- **Lazy Evaluation**: Iterables like generators produce elements on demand, optimizing memory usage.

An iterable in Python is any object that can return its elements one at a time, either through __iter__ or __getitem__. Iterables form the backbone of Python's data manipulation and looping mechanisms, providing a flexible and efficient way to handle sequences of data.

Generator Expressions

The Built-in Functions

Adapter Functions in Python

Adapter functions in Python are **higher-order functions** that transform or manipulate iterables without immediately consuming them. These functions return an **iterator** instead of a concrete data structure, allowing for **lazy evaluation** and efficient data processing. Unlike terminal functions, which produce a final result, adapter functions serve as intermediaries that modify the structure or contents of an iterable before it is eventually consumed.

Using adapter functions enables more expressive, modular, and performance-optimized code by applying transformations without requiring explicit loops. Some of the most commonly used adapter functions in Python include map(), filter(), zip(), sorted(), and enumerate(), each serving a unique purpose in functional programming.

map(function, iterable, ...)

The map() function applies a given function to each element in one or more iterables and returns an **iterator** of transformed values. It is useful when performing element-wise modifications without writing explicit loops.

Example: Squaring a List of Numbers

```
numbers = [1, 2, 3, 4, 5]

squared = map(lambda x: x ** 2, numbers)

print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

Here, map() applies the **lambda function** to each number, squaring the values lazily. The output is **materialized** into a list using list().

Example: Mapping Multiple Iterables

```
a = [1, 2, 3]

b = [4, 5, 6]

sums = map(lambda x, y: x + y, a, b)

print(list(sums))  # Output: [5, 7, 9]
```

Since map() supports multiple iterables, it applies the function pairwise to elements from a and b.

filter(function, iterable)

The filter() function **selects elements** from an iterable based on a predicate function that returns True or False. Only elements that satisfy the condition are retained.

Example: Filtering Even Numbers

```
numbers = [10, 15, 20, 25, 30]

evens = filter(lambda x: x % 2 == 0, numbers)

print(list(evens))  # Output: [10, 20, 30]
```

Here, filter() keeps only the numbers for which the lambda function evaluates to True (i.e., even numbers).

Example: Filtering Words by Length

```
words = ["apple", "banana", "kiwi", "grape"]

short_words = filter(lambda w: len(w) <= 5, words)

print(list(short_words))  # Output: ['apple', 'kiwi', 'grape']
```

This selects words with a length of **5 or fewer characters**.

zip(*iterables)

The zip() function combines multiple iterables element-wise, producing an **iterator of tuples** where each tuple contains corresponding elements from the input iterables.

Example: Pairing Elements from Two Lists

```
names = ["Alice", "Bob", "Charlie"]

scores = [85, 90, 78]

paired = zip(names, scores)

print(list(paired))  # Output: [('Alice', 85), ('Bob', 90),
('Charlie', 78)]
```

Here, zip() groups elements by index, creating tuples from corresponding elements in names and scores.

Example: Unzipping a Zipped List

```
pairs = [('Alice', 85), ('Bob', 90), ('Charlie', 78)]

names, scores = zip(*pairs)

print(names)  # Output: ('Alice', 'Bob', 'Charlie')

print(scores) # Output: (85, 90, 78)
```

Using *pairs, zip() **unzips** the tuples into separate sequences.

sorted(iterable, key=None, reverse=False)

The sorted() function returns a **sorted list** from an iterable, optionally using a key function to customize sorting behavior.

Example: Sorting Numbers

```
numbers = [5, 2, 8, 1, 9]

sorted_numbers = sorted(numbers)

print(sorted_numbers)  # Output: [1, 2, 5, 8, 9]
```

By default, sorted() arranges numbers in **ascending order**.

Example: Sorting with a Key Function

```
words = ["banana", "apple", "cherry"]

sorted_words = sorted(words, key=len)

print(sorted_words)  # Output: ['apple', 'banana', 'cherry']
```

Here, the key function len sorts words **by length**.

Example: Sorting in Descending Order

```
numbers = [5, 2, 8, 1, 9]

desc_sorted = sorted(numbers, reverse=True)

print(desc_sorted)  # Output: [9, 8, 5, 2, 1]
```

Setting reverse=True sorts the numbers in **descending order**.

enumerate(iterable, start=0)

The enumerate() function adds an **index** to an iterable, returning an **iterator of tuples** where each tuple contains an index and the corresponding element.

Example: Enumerating a List

```
fruits = ["apple", "banana", "cherry"]

indexed_fruits = enumerate(fruits)

print(list(indexed_fruits))

# Output: [(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

The function automatically assigns an index to each item, starting from **0**.

Example: Custom Start Index

```
fruits = ["apple", "banana", "cherry"]

indexed_fruits = enumerate(fruits, start=1)

print(list(indexed_fruits))

# Output: [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

By specifying start=1, the index begins from **1** instead of **0**.

Adapter functions are essential tools for transforming and restructuring iterables **without consuming them immediately**. They allow for **lazy evaluation**, meaning data is processed on demand, improving efficiency. By using functions like map(), filter(), zip(), sorted(), and enumerate(), Python programmers can write cleaner, more expressive code while avoiding explicit loops. These functions play a crucial role in functional programming, enabling more readable and efficient data transformations.

Terminal Functions

Terminal functions in Python are **higher-order functions** that consume an iterable and return a final result. Unlike adapter functions, which transform or modify iterables and return another iterable, terminal functions **collapse** data into a single value. They are often used to aggregate, filter, or summarize collections efficiently without requiring explicit loops.

These functions include **reduce(), any(), all(), max(), min(), and sum()**, each serving a different purpose in functional programming.

Using join() as a Terminal Function in Python

The join() method is a **terminal function** that consumes an iterable of strings and returns a single concatenated result. It is commonly used for formatting and combining strings efficiently.

Basic Usage

```
words = ["Hello", "world", "Python"]

sentence = " ".join(words)

print(sentence)  # Output: Hello world Python
```

Here, " ".join(words) concatenates the list of words with spaces in between.

Joining Non-String Iterables

join() only works on strings, so non-string iterables need to be converted first.

```
numbers = [1, 2, 3, 4]

result = "-".join(map(str, numbers))

print(result)  # Output: 1-2-3-4
```

Memory-Efficient with Generators

Using generators avoids creating intermediate lists.

```
words = ["apple", "banana", "cherry"]

print(", ".join(word for word in words if len(word) > 5))

# Output: banana, cherry
```

By avoiding multiple string concatenations (+), join() ensures faster, optimized, and memory-efficient string handling.

reduce(function, iterable, initializer=None)

The reduce() function from the functools module applies a **rolling computation** to elements in an iterable, reducing them to a single accumulated value. It takes a function that operates on two elements and **progressively applies it to the entire sequence**.

Example: Computing the Product of a List

```
from functools import reduce


numbers = [1, 2, 3, 4, 5]

product = reduce(lambda x, y: x * y, numbers)

print(product)  # Output: 120
```

Here, reduce() computes 1 * 2 * 3 * 4 * 5, reducing the list to 120.

Example: Using an Initializer

```
numbers = [1, 2, 3]

result = reduce(lambda x, y: x + y, numbers, 10)

print(result)  # Output: 16
```

With 10 as the initializer, the computation starts from 10 + 1 + 2 + 3 = 16.

reduce() is particularly useful when performing **cumulative calculations**, such as finding the sum, product, or greatest common divisor.

any(iterable)

The any() function **checks if at least one element in an iterable is truthy**, returning True if so and False otherwise.

Example: Checking for Any Non-Zero Value

```
values = [0, 0, 5, 0]

print(any(values))  # Output: True
```

Here, any() returns True because at least one value (5) is truthy.

Example: Checking for the Presence of a Condition

```
words = ["apple", "banana", "cherry"]

contains_vowel = any("a" in word for word in words)

print(contains_vowel)  # Output: True
```

Since "apple" and "banana" contain "a", any() returns True.

This function is commonly used for **short-circuit evaluation**, as it stops at the first True value.

all(iterable)

The all() function **checks if every element in an iterable is truthy**, returning True only if all elements evaluate to True.

Example: Checking if All Numbers Are Non-Zero

```
values = [1, 2, 3, 4]

print(all(values))  # Output: True
```

Since all numbers are nonzero, all() returns True.

Example: Checking if All Words Contain a Specific Letter

```
words = ["apple", "banana", "cherry"]

all_contain_a = all("a" in word for word in words)

print(all_contain_a)  # Output: True
```

Every word in the list contains "a", so all() returns True.

The all() function is useful for **validating data**, ensuring that all conditions hold before proceeding.

max(iterable, key=None) and min(iterable, key=None)

The max() and min() functions **return the largest and smallest elements** from an iterable. A **key function** can be provided to customize the comparison criteria.

Example: Finding the Maximum and Minimum Values in a List

```
numbers = [10, 25, 3, 45, 8]

print(max(numbers))  # Output: 45

print(min(numbers))  # Output: 3
```

Here, max() finds the largest number (45), while min() finds the smallest (3).

Example: Finding the Longest and Shortest Word

```
words = ["apple", "banana", "cherry"]

longest_word = max(words, key=len)

shortest_word = min(words, key=len)


print(longest_word)  # Output: banana

print(shortest_word) # Output: apple
```

The key=len argument sorts the words based on **length** instead of alphabetical order.

max() and min() are commonly used for **ranking data**, such as finding the best-performing student or the longest name in a dataset.

sum(iterable, start=0)

The sum() function **computes the sum of a sequence** of numbers, starting from an optional initial value.

Example: Summing a List of Numbers

```
numbers = [1, 2, 3, 4, 5]

total = sum(numbers)

print(total)  # Output: 15
```

Here, sum() adds all elements to return 15.

Example: Summing with an Initial Value

```
numbers = [1, 2, 3]

total = sum(numbers, 10)

print(total)  # Output: 16
```

With 10 as the starting value, sum() computes 10 + 1 + 2 + 3 = 16.

The sum() function is commonly used in **data aggregation**, such as computing total revenue or cumulative scores.

Functools and Itertools

Python provides **built-in functional programming capabilities** through **functional functions, functools, and itertools**, enabling developers to write concise, expressive, and efficient code. These tools facilitate **higher-order functions, lazy evaluation, function composition, and iterable manipulation**, making them essential for functional programming paradigms.

functools

The functools module **extends Python's functional programming capabilities**, providing decorators, memoization, and function composition tools.

Key Functionalities

- **Function Reduction**
    - **functools.reduce(function, iterable, initializer=None)** → Performs cumulative computation over an iterable.
- **Memoization and Caching**
    - **functools.lru_cache(maxsize=None)** → Caches function results to improve performance on repeated calls.
- **Function Composition and Partial Application**
    - **functools.partial(func, *args, **kwargs)** → Creates a new function with some arguments pre-filled.
    - **functools.wraps(wrapped, assigned, updated)** → Preserves metadata of wrapped functions in decorators.
- **Single Dispatch and Method Overloading**
    - **functools.singledispatch(func)** → Enables function overloading based on argument type.

Example: Using functools.reduce()

```
from functools import reduce


numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120
```

Here, reduce() applies multiplication cumulatively across all elements.

Example: Using functools.lru_cache() for Memoization

```
from functools import lru_cache


@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

```

```
print(fibonacci(10))  # Output: 55 (computed efficiently with caching)
```

Using **memoization** significantly speeds up recursive computations.

Example: Using functools.partial() for Function Pre-Filling

```
from functools import partial


def power(base, exponent):

    return base ** exponent


square = partial(power, exponent=2)

print(square(4))  # Output: 16
```

Here, partial() pre-fills the exponent argument, creating a **customized function**.

The functools module **enhances functional programming by providing tools for function optimization, composition, and higher-order behaviors**.

itertools

The itertools module provides **fast, memory-efficient iterators** for handling large or infinite sequences. These functions support **lazy evaluation**, reducing memory usage while enabling powerful data processing techniques.

Key Functionalities

- **Infinite Iterators**
  - **itertools.count(start=0, step=1)** → Generates an infinite sequence of numbers.
  - **itertools.cycle(iterable)** → Repeats an iterable indefinitely.
  - **itertools.repeat(object, times=None)** → Repeats a value indefinitely or a specified number of times.
- **Combinatoric Iterators**
  - **itertools.permutations(iterable, r=None)** → Generates all possible orderings of elements.
  - **itertools.combinations(iterable, r)** → Generates all subsets of length r.
  - **itertools.product(*iterables, repeat=1)** → Produces Cartesian products of input iterables.
- **Itertools for Efficient Data Processing**
  - **itertools.chain(*iterables)** → Combines multiple iterables into a single sequence.
  - **itertools.groupby(iterable, key=None)** → Groups adjacent elements based on a key function.

- **itertools.accumulate(iterable, func=operator.add)** → Performs cumulative computation.

Example: Using itertools.count() for Infinite Sequences

```
from itertools import count


for num in count(10, 2):  # Start at 10, increment by 2
    if num > 20:
        break
    print(num)
```

Output:

```
10
12
14
16
18
20
```

This generates numbers infinitely, stopping at 20 with a conditional check.

Example: Using itertools.combinations()

```
from itertools import combinations


items = ['A', 'B', 'C']
print(list(combinations(items, 2)))
# Output: [('A', 'B'), ('A', 'C'), ('B', 'C')]
```

This generates all unique pairs of elements.

Example: Using itertools.groupby() for Grouping Elements

```
from itertools import groupby


data = [(1, 'A'), (1, 'B'), (2, 'C'), (2, 'D')]
grouped = {k: list(v) for k, v in groupby(data, key=lambda x: x[0])}
print(grouped)
# Output: {1: [(1, 'A'), (1, 'B')], 2: [(2, 'C'), (2, 'D')]}
```

This groups elements by the first value in the tuple.

The itertools module enables **lazy, memory-efficient operations on iterators**, making it useful for **large datasets, combinatorics, and stream processing**.

Exercises

https://open.kattis.com/problems/addtwonumbers

https://open.kattis.com/problems/autori

https://open.kattis.com/problems/cold

https://open.kattis.com/problems/countthevowels

https://open.kattis.com/problems/greetings2

https://open.kattis.com/problems/jackolanternjuxtaposition

https://open.kattis.com/problems/jumbojavelin

https://open.kattis.com/problems/timeloop

https://open.kattis.com/problems/oddecho

https://open.kattis.com/problems/alphabetspam