**Functional Libraries in Python: functools and itertools**

<https://docs.python.org/3/library/functools.html>

Python provides several libraries to facilitate functional programming. Two of the most commonly used libraries are functools and itertools. These libraries offer a variety of functions and tools to enhance the functional programming capabilities in Python.

**functools**

The functools module is a standard library module for higher-order functions: functions that act on or return other functions. Some of the key features of functools include function caching, partial function application, and composition.

**Key Functions in functools**

1. **functools.reduce()**

Applies a function cumulatively to the items of an iterable, reducing it to a single value.

from functools import reduce

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

sum\_of\_numbers = reduce(lambda x, y: x + y, numbers)

print(sum\_of\_numbers) # Output: 15

1. **functools.partial()**

Creates a new function with some arguments of the original function fixed.

from functools import partial

def multiply(x, y):

return x \* y

double = partial(multiply, 2)

print(double(5)) # Output: 10

1. **functools.lru\_cache()**

Decorator to cache the results of a function, improving performance for expensive or I/O-bound functions.

from functools import lru\_cache

@lru\_cache(maxsize=32)

def fibonacci(n):

if n < 2:

return n

return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10)) # Output: 55

1. **functools.wraps()**

A decorator to help write decorators that preserve the original function's metadata.

from functools import wraps

def my\_decorator(f):

@wraps(f)

def wrapper(\*args, \*\*kwargs):

print("Something is happening before the function is called.")

result = f(\*args, \*\*kwargs)

print("Something is happening after the function is called.")

return result

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

1. **functools.singledispatch()**

A decorator to transform a function into a single-dispatch generic function, allowing you to define multiple implementations based on the type of the first argument.

from functools import singledispatch

@singledispatch

def process(arg):

print("Default implementation")

@process.register(int)

def \_(arg):

print("Processing an integer")

@process.register(str)

def \_(arg):

print("Processing a string")

process(42) # Output: Processing an integer

process("hello") # Output: Processing a string

process([1, 2, 3]) # Output: Default implementation

**itertools**

The itertools module provides a collection of tools for handling iterators. It includes functions for creating iterators for efficient looping, making it possible to work with infinite sequences, combinatorics, and more.

**Key Functions in itertools**

1. **itertools.count()**

Creates an iterator that returns evenly spaced values starting from a specified number.

from itertools import count

for i in count(10, 2):

print(i)

if i >= 20:

break

1. **itertools.cycle()**

Creates an iterator that returns elements from the iterable and saves a copy of each. When the iterable is exhausted, it returns elements from the saved copy indefinitely.

from itertools import cycle

counter = 0

for item in cycle(['A', 'B', 'C']):

print(item)

counter += 1

if counter == 10:

break

1. **itertools.repeat()**

Creates an iterator that returns the same value over and over again.

from itertools import repeat

for item in repeat('Hello', 3):

print(item)

1. **itertools.chain()**

Creates an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted.

from itertools import chain

list1 = [1, 2, 3]

list2 = [4, 5, 6]

combined = chain(list1, list2)

print(list(combined)) # Output: [1, 2, 3, 4, 5, 6]

1. **itertools.islice()**

Creates an iterator that returns selected elements from the iterable. Works like slicing a list but returns an iterator.

**Syntax:**

itertools.islice(iterable, start, stop[, step])

* iterable: The input iterable.
* start: The starting index (inclusive).
* stop: The stopping index (exclusive).
* step (optional): The step between elements (default is 1).

from itertools import islice

items = range(10)

sliced = islice(items, 2, 8, 2)

print(list(sliced)) # Output: [2, 4, 6]

1. **itertools.combinations()**

Creates an iterator that returns all possible combinations of a specified length from the input iterable.

from itertools import combinations

items = ['A', 'B', 'C']

comb = combinations(items, 2)

print(list(comb)) # Output: [('A', 'B'), ('A', 'C'), ('B', 'C')]

1. **itertools.permutations()**

Creates an iterator that returns all possible permutations of a specified length from the input iterable.

from itertools import permutations

items = ['A', 'B', 'C']

perm = permutations(items, 2)

print(list(perm)) # Output: [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

**Summary**

* **functools**: A module for higher-order functions, offering tools like reduce(), partial(), lru\_cache(), wraps(), and singledispatch().
* **itertools**: A module for creating and working with iterators, providing functions like count(), cycle(), repeat(), chain(), islice(), combinations(), and permutations().

These libraries enhance Python's functional programming capabilities, making it easier to write concise and efficient code for common tasks involving iterables and higher-order functions.