Python: Generators

- Video: How to Use Yield to Generate Values
- Video: Learn to Generate the Fibonacci Sequence in Python using Generators

What are Generators?

Generators are special types of functions that use the `yield` keyword to produce a series of values, rather than computing them all at once and returning them in a list, for example.

Why Use Generators?

1. *Memory Efficiency*: Generators use significantly less memory than storing all values in a list.
2. *Lazy Evaluation*: Generators only compute values when they're needed.
3. *Improved Performance*: Generators can improve performance by avoiding unnecessary computations.

Basic Syntax

```
def generator_function_name(parameters):
    # Some code here
    yield expression
    # More code can follow
```

Consider a simple example where we want to generate the squares of the first five natural numbers:

```
def generate_squares(n):
    for i in range(n):
        yield i ** 2
```

Using the generator

```
squares = generate_squares(5)
for square in squares:
    print(square) # Output: 0, 1, 4, 9, 16
```

In this example, when generate_squares(5) is called, it returns a generator object. The for loop then iterates over this object. Each time the loop requests the next value, the generator function resumes execution from where it last left off (after the yield statement), calculates the next square, and yields it. This process continues until the loop finishes or the generator runs out of values to yield .

This on-demand generation of values is known as lazy evaluation . Importantly, the state of the function (including the value of i) is preserved between calls to yield . This means the function can pick up exactly where it left off, making it efficient for processing sequences step by step.

Example:

```python
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1

gen = infinite_sequence()
print(next(gen))  # prints 0
print(next(gen))  # prints 1
print(next(gen))  # prints 2
```

In this example, `infinite_sequence` is a generator that produces an infinite sequence of numbers. The `next()` function is used to retrieve the next value from the generator.

Introducing Generator Expressions

Python also offers a more concise way to create generators using generator expressions . These are similar to list comprehensions but use parentheses () instead of square brackets [] .

Here's the syntax for a generator expression: (expression for item in iterable if condition)

Let's rewrite our previous example using a generator expression:

```python
squares = (i ** 2 for i in range(5))
for square in squares:
    print(square) # Output: 0, 1, 4, 9, 16
```

The output is the same, but the syntax is more compact. The key difference between a list comprehension and a generator expression lies in what they produce. A list comprehension creates the entire list in memory at once, whereas a generator expression returns a generator object that yields items one at a time . This makes generator expressions particularly useful when dealing with large or potentially infinite sequences, as they avoid the memory overhead of storing the entire sequence .

for more details, see Python Generators: A Beginner's Guide to Memory-Efficient Iteration

Key Terms

True/False (Mark T for True and F for False)

Answer Key (True/False):

Multiple Choice (Select the best answer)

1. **Which function would you use to determine the type of a variable in Python?**
    - A) id()
    - B) type()
    - C) str()
    - D) isinstance()

What is the correct way to create a generator in Python? a) def generator(): b) yield generator() c) generator = () d) generator = []

Answer: b) yield generator()

Watch this video for the answer:

Answer key (Mutiple Choice):

Fill in the Blanks

Answer Key (Fill in the Blanks):

Exercises

Review Questions

References and Bibliography