TOPS Technology

Python Fundamentals

Presented By:

Nandni Vala

Generators and Iterators

1.Understanding how generators work in Python.

- A **generator** is a special type of iterable in Python that allows you to produce values one at a time using the yield keyword, instead of returning all values at once. This makes generators memory-efficient and suitable for handling large datasets.
- **Example**:
- def simple_generator():
-     yield 1
-     yield 2
-     yield 3
-
- gen = simple_generator()
- print(next(gen))
- print(next(gen))
- print(next(gen))

2.Difference between yield and return.

Feature	yield	return
Definition	Produces a value and pauses the function without terminating it.	Ends the function and returns a value immediately.
Usage	Used in generator functions to create iterators.	Used in regular functions to return a single value or result.
State Preservation	Preserves the function's execution state, allowing it to resume from where it paused.	Does not preserve state; the function terminates upon execution.
Number of Values	Can yield multiple values over time.	Returns a single value and terminates.
Type of Function	Used in generator functions.	Used in standard functions.
Execution	Executes lazily, producing values one at a time when requested.	Executes eagerly, completing all operations before returning.

3.Understanding iterators and creating custom iterators.

- An **iterator** is an object that allows sequential traversal of elements in a collection without exposing the internal structure.
- **Iterable**: An object that can return an iterator using __iter__().
- **Iterator**: An object with __next__() to fetch the next item; raises StopIteration when items are exhausted.
- **Example**:
- nums = [1, 2, 3]
- it = iter(nums)
- print(next(it))
- print(next(it))

- **Creating Custom Iterators**
- To create a custom iterator, define a class with __iter__() and __next__() methods.
- **Example**:

```
class MyIterator:
    def __init__(self, max_value):
        self.current = 0
        self.max_value = max_value

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.max_value:
            self.current += 1
            return self.current
        else:
            raise StopIteration
# Usage
for num in MyIterator(3):
    print(num)
```